Programming in
VAX-11 PL/I

AA-L057B-TE

November 1983

This manual provides an informal introduction and usage guide for the VAX-11 PL/I programming language.

digital equipment corporation · maynard, massachusetts

First Printing, July 1981
Revised, November 1983

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.

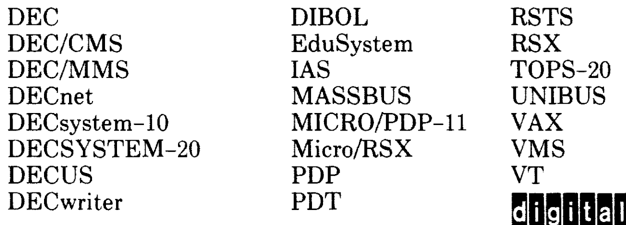
No responsibility is assumed for the use or reliability of software on equipment that is not supplied by Digital Equipment Corporation or its affiliated companies.

Copyright © 1981, 1983 by Digital Equipment Corporation
All Rights Reserved.

Printed in U.S.A.

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

The following are trademarks of Digital Equipment Corporation:



DEC	DIBOL	RSTS
DEC/CMS	EduSystem	RSX
DEC/MMS	IAS	TOPS-20
DECnet	MASSBUS	UNIBUS
DECsystem-10	MICRO/PDP-11	VAX
DECSYSTEM-20	Micro/RSX	VMS
DECUS	PDP	VT
DECwriter	PDT	digital

ZK2380

HOW TO ORDER ADDITIONAL DOCUMENTATION

In Continental USA and Puerto Rico call 800-258-1710

In New Hampshire, Alaska, and Hawaii call 603-884-6660

In Canada call 613-234-7726 (Ottawa-Hull)
800-267-6146 (all other Canadian)

DIRECT MAIL ORDERS (USA & PUERTO RICO)*

Digital Equipment Corporation
P.O. Box CS2008
Nashua, New Hampshire 03061

*Any prepaid order from Puerto Rico must be placed with the local Digital subsidiary (809-754-7575)

DIRECT MAIL ORDERS (CANADA)

Digital Equipment of Canada Ltd.
940 Belfast Road
Ottawa, Ontario K1G 4C2
Attn: A&SG Business Manager

DIRECT MAIL ORDERS (INTERNATIONAL)

Digital Equipment Corporation
A&SG Business Manager
c/o Digital's local subsidiary or approved distributor

Internal orders should be placed through the Software Distribution Center (SDC), Digital Equipment Corporation, Northboro, Massachusetts 01532

Contents

Page

Preface xiii

Chapter 1 Introduction to Program Development on VAX/VMS

1.1 VAX/VMS Commands for Program Development 1
1.1.1 Hints for Entering Commands 4
1.1.2 HELP 4
1.2 File Specifications and Defaults 5
1.2.1 Directories and Subdirectories 8
1.2.2 Changing the Default Directory 8
1.2.3 Logical Names 8
1.3 File Creation and Maintenance 10
1.4 Command Procedures 13
1.4.1 Command Procedures for Program Development 13
1.4.2 The Login Command Procedure File 16

Chapter 2 Creating and Correcting Programs

2.1 Introduction to EDT 18
2.1.1 Line Editing Command Summary 19
2.1.2 The Help Facilities 21
2.2 Invoking and Terminating EDT 22
2.2.1 Invoking EDT 22
2.2.2 Terminating EDT 23
2.3 Creating a New File in Line Mode 24
2.4 Editing an Existing File in Line Mode 24
2.4.1 Range Specifications 25
2.4.2 Maneuvering in the File 28
2.4.3 Inserting New Text 29
2.4.4 Deleting and Replacing Text 30
2.4.5 Moving Text 31
2.4.6 Substituting Text 31
2.4.7 Input From and Output To Files 33
2.4.8 Editing a File From Another Directory 33
2.5 Character Editing 34
2.5.1 Entering and Exiting Character Editing Mode 35
2.5.2 Maneuvering the Cursor 35
2.5.3 Inserting Text 38
2.5.4 Deleting and Undeleting Text 38
2.5.5 Moving Text 39

2.6 Protecting and Recovering Text 40
2.7 EDT Aids for the Programmer 41
    2.7.1 Structured Tabs 41
    2.7.2 Special-Purpose Key Definitions 42
    2.7.3 Startup Command Files 43

Chapter 3 Compiling PL/I Programs

3.1 Functions of the Compiler. 45
3.2 The PLI Command 46
    3.2.1 PLI Command Examples 52
    3.2.2 Specifying Input and Output Files 53
3.3 Using Text Libraries 53
    3.3.1 Specifying Text Libraries in the PLI Command 54
    3.3.2 Default PL/I Libraries 55
3.4 Compiler Diagnostic Messages and Error Conditions 55
3.5 User-Generated Diagnostic Messages 57

Chapter 4 Linking Programs

4.1 Functions of the Linker 59
4.2 Using the LINK Command 59
    4.2.1 Linker Messages 60
    4.2.2 Linker Input Files 62
    4.2.3 Linker Output Files 62
4.3 Using Object Module Libraries 63
    4.3.1 Defining the Search Order for Libraries 63
    4.3.2 Default User Object Module Libraries. 64
    4.3.3 Temporary Defaults for INCLUDE Files. 65
    4.3.4 System Libraries 65
    4.3.5 Creating Shareable Images 66

Chapter 5 Running PL/I Programs on VAX/VMS

5.1 The RUN Command 67
    5.1.1 Image Exit. 67
    5.1.2 Run-Time Errors 68
    5.1.3 Interrupting a Program 69
5.2 Returning Status Values to the Command Interpreter 70

Chapter 6 Creating Libraries

6.1 The LIBRARY Command. 72
6.2 Creating and Correcting Text Libraries 77
6.3 Creating and Correcting Object Module Libraries 78

Chapter 7 Program Structure and Content

7.1 Blocks 82
    7.1.1 Begin Blocks 82
    7.1.2 Procedures 83
7.2 Statements 84
    7.2.1 Statement Labels 84
    7.2.2 Keywords and Punctuation 84

7.2.3 Identifiers 86
7.2.3.1 Rules for Identifiers 86
7.2.4 Alphabetic Summary of Statements 87
7.3 Data and Variables 88
7.4 Program Text 91
7.4.1 Program Format 92
7.4.2 Comments 92
7.4.3 %INCLUDE Statement 93

Chapter 8 Data Types

8.1 Summary of Data Types 95
8.2 Arithmetic Data Types 96
8.2.1 Fixed-Point Binary Data 97
8.2.2 Fixed-Point Decimal Data 98
8.2.2.1 Fixed-Point Decimal Constants 98
8.2.2.2 Fixed-Point Decimal Variables 99
8.2.2.3 Using Fixed-Point Data in Expressions 99
8.2.3 Floating-Point Data 100
8.2.3.1 Constants 100
8.2.3.2 Variables. 101
8.2.3.3 Using Floating-Point Data in Expressions 101
8.2.3.4 G_FLOAT and H_FLOAT Support 102
8.2.3.5 Floating-Point Data Formats 102
8.2.4 Pictured Data 103
8.2.4.1 Pictured Variables 103
8.2.4.2 Assigning Values to Pictured Variables 104
8.2.4.3 Extracting Values from Pictured Data 105
8.2.4.4 Picture Characters 106
8.2.5 Precision of Arithmetic Data Types 111
8.2.6 Scale of Fixed-Point Data Types 112
8.3 Character-String Data 113
8.3.1 Character-String Constants 113
8.3.2 Character-String Variables 114
8.3.2.1 Fixed-Length Character-String Variables 114
8.3.2.2 Varying-Length Character-String Variables 115
8.4 Bit-String Data 115
8.4.1 Bit-String Constants 117
8.4.2 Bit-String Variables 118
8.4.3 Alignment of Bit-String Data 119
8.4.4 Bit Strings and Integers 119
8.4.5 Replication Factor for String Constants 120

Chapter 9 Storage Classes

9.2 Automatic Variables 123
9.2 Static Variables 123
9.3 Internal Variables 124
9.4 External Variables 124
9.5 Based Variables 126
9.5.1 Data Types Used With Based Variables 126
9.5.1.1 Pointer Data 127
9.5.1.2 Area and Offset Data 127

9.5.2 Declaring Based Variables 129
9.5.3 ALLOCATE Statement 130
9.5.4 FREE Statement 131
9.5.5 Referring to Based Variables 131
9.5.6 Example of Based Variable Use 135
9.5.7 Data Type Matching for Based Variables 136
9.5.7.1 Matching by Overlay Defining 137
9.5.7.2 Matching by Left-to-Right Equivalence 137
9.5.7.3 Nonmatching Based Variable References 138
9.6 Controlled Variables 138
9.6.1 Using the ALLOCATION Built-In Function 140
9.6.2 Using the ADDR Built-In Function 140
9.7 Defined Variables 141

Chapter 10 Aggregates

10.1 Arrays 144
10.1.1 Array Declarations 144
10.1.2 References to Individual Elements 146
10.1.3 Initializing Arrays 147
10.1.4 Assigning Values to Array Variables 148
10.1.5 Order of Assignment and Output for Multidimensional Arrays . 149
10.2 Structures 149
10.2.1 Structure Declarations 150
10.2.2 Member Attributes 152
10.2.2.1 Using the LIKE Attribute 152
10.2.2.2 Using the REFER Option 153
10.2.2.3 Using the UNION Attribute 156
10.2.3 Structure-Qualified References 158
10.2.4 Arrays of Structures 160
10.2.4.1 Arrays of Structures that Contain Arrays 160
10.2.4.2 Connected and Unconnected Arrays 161

Chapter 11 Declarations

11.1 Declare Statement 163
11.1.1 Simple Declarations 164
11.1.2 Multiple Declarations 164
11.1.3 Factored Declarations 165
11.1.4 Declarations Outside of Procedures 166
11.1.5 Initializing Variables in the DECLARE Statement 166
11.2 Scope of Declarations 168

Chapter 12 Expressions and Assignments

12.1 Assignment Statement 171
12.2 Operators and Operands 173
12.2.1 Operators 174
12.2.2 Operands 175
12.3 Expression Evaluation and Precedence of Operations 175
12.4 Conversion of Operands and Expressions 177
12.4.1 Derived Data Types for Arithmetic Operations 177
12.4.2 Built-In Conversion Functions 178
12.4.3 Implicit Conversion During Assignment 179
12.5 Pseudovariables 180

Chapter 13 Procedures

13.1 Using Procedures 187
13.1.1 Procedure Usage Concepts 188
13.1.1.1 Entry Points 188
13.1.1.2 Passing Arguments to Subroutines and Functions. . . 189
13.1.1.3 Terminating Procedures 189
13.1.2 PROCEDURE Statement. 190
13.1.3 ENTRY Statement. 191
13.1.4 CALL Statement. 193
13.1.5 Functions and Function References 194
13.1.6 RETURN Statement 195
13.1.7 RETURNS Attribute and Option 196
13.1.8 Parameters and Arguments 198
13.1.8.1 Rules for Specifying Parameters 198
13.1.8.2 Argument Passing. 200
13.2 Calling External Procedures 202
13.2.1 Entry Data 204
13.2.1.1 Entry Constants. 204
13.2.1.2 Entry Variables 206
13.2.2 Passing Arguments to Non-PL/I Procedures 206
13.2.2.1 Passing Arguments by Immediate Value 206
13.2.2.2 Passing Arguments by Reference. 207
13.2.2.3 Passing Arguments by Descriptor 208

Chapter 14 Program Control

14.1 DO Statement. 210
14.1.1 Simple DO. 210
14.1.2 DO WHILE 211
14.1.3 DO UNTIL. 212
14.1.4 Controlled DO 213
14.1.5 DO REPEAT. 215
14.2 BEGIN Statement. 216
14.3 END Statement 217
14.4 IF Statement 218
14.5 SELECT Statement 220
14.6 GOTO Statement 222
14.6.1 Label Array Constants 223
14.6.2 Label Variables 224
14.7 LEAVE Statement. 224
14.8 STOP Statement 226
14.9 NULL Statement 226

Chapter 15 Error Handling

15.10 ON Statement 227
15.1.1 Contents of an ON-Unit. 230
15.1.2 Search for ON-Units 231
15.1.3 Completion of ON-Units. 232

15.1.4 ON Condition Descriptions . . . 232
15.1.5 ON-Unit Examples . . . 240
15.2 REVERT Statement . . . 241
15.3 SIGNAL Statement . . . 242
15.4 Resignal Built-In Subroutine . . . 242

Chapter 16 File Control

16.1 File Control Statements . . . 243
16.1.1 Declaring a File . . . 243
16.1.2 OPEN Statement . . . 244
16.1.2.1 General-Purpose Attributes and Options . . . 246
16.1.2.2 Opening a File . . . 247
16.1.3 CLOSE Statement . . . 250
16.2 PL/I Files and VAX/VMS File Specifications . . . 251
16.2.1 The TITLE Option . . . 251
16.2.2 Using Logical Names . . . 252
16.2.3 Process Permanent Logical Names . . . 253
16.2.4 Expanding File Specifications . . . 254
16.3 Summary of Environment Options . . . 255
16.4 Summary of File-Handling Built-In Subroutines . . . 263
16.4.1 DISPLAY Built-In Subroutine . . . 263
16.4.2 EXTEND Built-In Subroutine . . . 268
16.4.3 FLUSH Built-In Subroutine . . . 271
16.4.4 NEXT_VOLUME Built-In Subroutine . . . 271
16.4.5 REWIND Built-In Subroutine . . . 272
16.4.6 SPACEBLOCK Built-In Subroutine . . . 272
16.5 File Error Handling . . . 273
16.5.1 Values Returned by PL/I Built-In Functions . . . 273
16.5.2 Writing an Error Handler . . . 274
16.5.3 Default Error Handling . . . 275

Chapter 17 Stream Input/Output

17.1 Statements for Stream I/O . . . 278
17.1.1 GET Statement . . . 278
17.1.1.1 Common Syntax Elements . . . 279
17.1.1.2 GET EDIT . . . 280
17.1.1.3 GET LIST . . . 282
17.1.1.4 GET SKIP . . . 284
17.1.2 PUT Statement . . . 285
17.1.2.1 Common Syntax Elements . . . 285
17.1.2.2 PUT EDIT . . . 288
17.1.2.3 PUT LINE . . . 289
17.1.2.4 PUT LIST . . . 290
17.1.2.5 PUT PAGE . . . 291
17.1.2.6 PUT SKIP . . . 292
17.1.3 FORMAT Statement . . . 292
17.2 Stream I/O Processing and Positioning . . . 292
17.2.1 Processing and Positioning of Stream Files . . . 293
17.2.2 Processing and Positioning of Print Files . . . 294
17.2.3 Processing and Positioning of Character Strings . . . 296

17.3 Format Items and Specifications 296
17.3.1 Format Items 296
17.3.2 Format Specifications 297

Chapter 18 Record Input/Output

18.1 Statements for Record I/O 301
18.1.1 READ Statement 304
18.1.2 WRITE Statement 308
18.1.3 REWRITE Statement 311
18.1.4 DELETE Statement 314
18.1.5 Options for Record I/O Statements 315
18.2 Sequential Files 318
18.2.1 Creating a Sequential File 318
18.2.2 Using Magnetic Tape Files 319
18.2.2.1 Format of Magnetic Tapes. 320
18.2.2.2 Multivolume Tape Files 320
18.3 Relative Files 321
18.3.1 The Organization of a Relative File 321
18.3.2 Creating a Relative File 321
18.3.2.1 Maximum Record Number 323
18.3.2.2 Maximum Record Size 323
18.3.3 Using Relative Files 323
18.3.3.1 Updating a Relative File. 325
18.3.3.2 Reading a Relative File Sequentially 325
18.3.3.3 Error Handling 326
18.4 Indexed Sequential Files 326
18.4.1 Indexed File Organization 327
18.4.2 Defining an Indexed Sequential File. 329
18.4.3 Using Indexed Sequential Files 330
18.4.3.1 Reading an Indexed Sequential File Sequentially . . . 331
18.4.3.2 Accessing Records by Alternate Key 332
18.4.3.3 Updating Records in an Indexed Sequential File . . . 333
18.4.3.4 Error Handling 333

Chapter 19 Built-In Functions

19.1 Summary of Built-In Functions 334
19.2 Built-In Function Descriptions 338

Chapter 20 Compile Time Facilities

20.1 The VAX–11 Common Data Dictionary(CDD) 372
20.1.1 Creating and Maintaining a CDD. 373
20.1.2 Using the CDD. 374
20.2 The VAX–11 PL/I Embedded Preprocessor 376
20.2.1 Preprocessor Compilation Control. 376
20.2.2 Preprocessor Procedures 377
20.2.3 Preprocessor Statements 380
20.2.3.1 %Assignment Statement. 382
20.2.3.2 %Null Statement 382
20.2.3.3 %ACTIVATE Statement 382

20.2.3.4 %DEACTIVATE Statement 384
20.2.3.5 %DECLARE Statement 385
20.2.3.6 %DICTIONARY Statement 386
20.2.3.7 %DO Statement. 387
20.2.3.8 %END Statement. 387
20.2.3.9 %ERROR Statement 388
20.2.3.10 %FATAL Statement 388
20.2.3.11 %GOTO Statement 388
20.2.3.12 %IF Statement. 389
20.2.3.13 %INCLUDE Statement. 390
20.2.3.14 %INFORM Statement 391
20.2.3.15 %LIST Statement 391
20.2.3.16 %NOLIST Statement 392
20.2.3.18 %PAGE Statement. 392
20.2.3.19 %PROCEDURE Statement. 392
20.2.3.20 %REPLACE Statement 396
20.2.3.21 %RETURN Statement 396
20.2.3.22 %SBTTL Statement 397
20.2.3.23 %TITLE Statement 397
20.2.3.24 %WARN Statement 398
20.2.4 Preprocessor Built-In Functions 398
20.2.4.1 ERROR Preprocessor Built-In Function 400
20.2.4.2 INFORM Preprocessor Built-In Function. 400
20.2.4.3 LINE Preprocessor Built-In Function. 400
20.2.4.4 VARIANT Preprocessor Built-In Function 400
20.2.4.5 WARN Preprocessor Built-In Function 401

Appendix A Rules for Conversion of Data

A.1 Assignments to Arithmetic Variables 402
A.1.1 Arithmetic to Arithmetic Conversions 402
A.1.2 Pictured to Arithmetic Conversions. 402
A.1.3 Bit-String to Arithmetic Conversions 403
A.1.4 Character String to Arithmetic Conversions. 403
A.2 Assignments to Bit-String Variables. 404
A.2.1 Arithmetic and Pictured to Bit-String Conversions 404
A.2.2 Character-String to Bit-String Conversions 404
A.3 Assignments to Character-String Variables 405
A.3.1 Arithmetic to Character-String Conversions. 405
A.3.1.1 Conversion from Fixed-Point Binary or Decimal 405
A.3.1.2 Conversion from Floating-Point Binary or Decimal . . . 406
A.3.2 Pictured to Character-String Conversions. 407
A.3.3 Bit-String to Character-String Conversions 408
A.4 Assignments to Pictured Variables 408
A.5 Conversions between Offsets and Pointers 408

Appendix B Calling System Services

B.1 Declaring System Services 410
B.2 SPECIFYING ARGUMENTS FOR SYSTEM SERVICES 411
B.2.1 Argument-Passing Mechanisms Used by System Services 411
B.2.2 Parameter Descriptors for System Services Data Types 412
B.2.3 Variable-Length Argument Lists 413
B.2.4 Symbol Definitions for System Service Arguments 413

B.3 Testing Return Values from System Services 414
B.4 Examples of System Services 415
B.4.1 Logical Name Translation 415
B.4.2 Timer and Time Conversion Routines 417
B.4.2.1 Obtaining a Time Value in System Format 417
B.4.2.2 Setting the Timer. 418

Appendix C ASCII Character Set

Index

Figures

1 Commands for PL/I Program Development 2
2–1 VT52 Keypad 36
2–2 VT100 Keypad 36
6–1 Creating and Using an INCLUDE File Library 78
6–2 Creating and Using an Object Module Library 80
7–1 Using the %INCLUDE Statement 93
9–1 External Variables. 125
9–2 Using the ADDR Built-In Function 134
9–3 An Overlay Defined Variable. 143
10–1 Connected and Unconnected Arrays 162
11–11 Scope of Internal Names 169
15–1 Search for ON-Units 233
17–1 Forms of the GET Statement 278
17–2 Forms of the PUT Statement 286
18–1 A Relative File 322
18–2 An Indexed Sequential File 328
19–1 Example of the BOOL Built-In Function. 343

Tables

1–1 Summary of File Specification Syntax 6
1–2 Commands for Maintaining Logical Names. 10
1–3 VAX/VMS Commands for File Maintenance 11
2–1 Summary of Line Editing Commands 20
2–2 Single-Line Range Specifications. 25
2–3 Multiple-Line Range Specifications 26
3–1 PL/I Compiler Options 50
3–2 Listing Notation Characters 51
4–1 LINK Command Qualifiers 61
7–1 Punctuation Marks Recognized by VAX–11 PL/I 85
7–2 Summary of VAX–11 PL/I Statements 87
7–3 Summary of VAX–11 PL/I Attributes 89
8–1 Implied Attributes for Computational Data. 97
8–2 VAX–11 Floating-Point Types 102
8–3 Floating-Point Types Used by PL/I. 103
8–4 Picture Characters. 106
8–5 ASCII Representation of Encoded-Sign Digits 109
12–1 Operators 174
12–2 Precedence of Operations 176
12–3 Built-In Functions for Conversions Between Arithmetic and Nonarithmetic Types. 179

15–1 Summary of ON Conditions 228
16–1 File Description Attributes 245
16–2 File Description Attributes Implied at Open Time 248
16–3 Default Process Logical Names 254
16–4 Summary of ENVIRONMENT Options 256
16–5 Summary of File-Handling Built-In Subroutines 263
16–6 ENVIRONMENT Option Values Returned by DISPLAY 265
16–7 File Attribute Information Returned by DISPLAY 269
16–8 Device Information Returned by DISPLAY 270
17–1 Summary of Format Items 298
18–1 Attributes and Access Modes for Record Files 302
18–2 Key Data Types 332
19–1 Summary of PL/I Built-In Functions 335
20–1 Summary of PL/I Preprocessor Statements 380
20–2 Summary of PL/I Preprocessor Built-In Functions 399
B–1 Input Arguments for System Services 412
B–2 Output Arguments for System Services 413
C–1 ASCII Character Set 422

Preface

This manual is...

An informal guide for programmers who want to write PL/I programs that will be executed on a VAX–11 computer running the VMS operating system. It provides general information on creating, correcting, and compiling PL/I programs, as well as reference information on the language and its syntax rules.

Readers of this manual are assumed to have prior knowledge and understanding of the PL/I language. From this manual, they can determine what statements, data type attributes, and built-in functions are available in VAX–11 PL/I and can get started writing new programs or modifying existing programs.

This manual is not...

A complete and detailed reference manual for the VAX–11 PL/I language nor for the VAX/VMS command language.

Where to Find More Information

Introduction to VAX–11 PL/I contains an overview of the PL/I language and its implementation for the VAX–11 computer. The *Introduction* is recommended for all programmers who are not familiar with PL/I or who need information on the VAX–11-specific capabilities of VAX–11 PL/I.

The *VAX–11 PL/I Encyclopedic Reference* contains a complete definition of the VAX–11 PL/I programming language, including the keywords and the semantic and syntax rules of PL/I statements, attributes, and built-in functions. The *Encyclopedic Reference* contains descriptions of language elements and topics in alphabetic order.

The companion document to the *Encyclopedic Reference* is the *VAX–11 PL/I User's Guide*. It contains information on developing programs with the VAX/VMS command language, on using the extensive I/O capabilities provided in VAX–11 PL/I, and on programming facilities available to PL/I programs executing under the exclusive control of the VAX/VMS operating system.

The manuals that accompany the operating system provide full information about VAX/VMS; this manual makes reference to some of them. For a complete list of all VAX/VMS documents and their order numbers, see the *VAX-11 Information Directory and Index.*

Summary of Contents

Programming in VAX-11 PL/I consists of 20 chapters and 3 appendixes:

- Chapter 1 introduces the procedures you use to create, test, and correct a PL/I program on the VAX/VMS operating system.
- Chapter 2 provides information you need to create and correct a PL/I source file. It describes EDT, the DEC Standard Editor.
- Chapter 3 describes the VAX-11 PL/I compiler and the command that invokes it.
- Chapter 4 describes the linker, which converts the output of the compiler into an executable image that you can run on the system.
- Chapter 5 describes the RUN command, which starts program execution.
- Chapter 6 describes the library utility, which you can use to create text and object module libraries that contain code common to several programs.
- Chapter 7 describes the structure of a PL/I program and summarizes the various elements of a program. Chapter 7 introduces the rest of this manual, which covers the PL/I language in greater detail.
- Chapter 8 describes PL/I's computational data types.
- Chapter 9 describes the various storage classes to which a variable can belong.
- Chapter 10 describes aggregates, which are collections of variables that can be referred to by name or individually.
- Chapter 11 describes data declarations.
- Chapter 12 describes expressions and assignments, which are used by PL/I programs to compute values and assign them to variables.
- Chapter 13 describes how you can use procedures as subroutines or functions.
- Chapter 14 describes statements and a data type that you can use to control the flow of execution within your program.
- Chapter 15 describes statements and a built-in subroutine that allow your program to respond to errors that occur during execution.

- Chapter 16 describes PL/I statements, data types, and built-in subroutines that allow your program to control files.
- Chapter 17 describes statements and techniques for performing stream I/O, which consists of a stream of characters passed to or from the program.
- Chapter 18 describes statements and techniques for performing record I/O, in which the program reads and writes entire records instead of streams of characters.
- Chapter 19 describes the PL/I built-in functions, which provide a variety of services to the programmer.
- Chapter 20 describes the VAX–11 PL/I compile-time facilities, which permit conditional compilation.
- Appendix A contains the rules that PL/I follows when it converts values of one data type to another data type.
- Appendix B describes how a PL/I program can call a VAX/VMS system service to perform system-specific operations not available through PL/I statements.
- Appendix C contains a table of the set of ASCII characters.

Conventions Used in This Document

RET	A symbol with a 1- to 3-character abbreviation indicates that you press a key on the terminal, for example, RET or ESC.
CTRL/x	The symbol CTRL/x indicates that you press the key "x" while holding down the key labeled CTRL, for example, CTRL/C. In examples, this control key sequence is shown as ^x, for example ^C, because that is how the VAX/VMS system prints control key sequences.
`Enter string>Abcd` RET	In computer dialogs, the user's response to a prompt is printed in red ink.

`DECLARE X FIXED;`
`.`
`.`
`.`
`X = 5 ;`
	A vertical ellipsis indicates that not all of the text of a program or program output is illustrated. Only relevant material is shown.

option,...	A horizontal ellipsis indicates that additional parameters, options, or values can be entered. A comma preceding the ellipsis indicates that successive items must be separated by commas.

quotation mark apostrophe	The term quotation mark refers to the quotation mark symbol ("). The term apostrophe refers to the single quotation mark symbol (′).
[OPTIONS (option,...)]	Except in VMS file specifications, square brackets indicate that a syntactic element is optional.
[LIST EDIT]	Brackets surrounding two or more stacked items indicate conflicting options, one of which *may* be chosen.
{ /ALL module,... }	Braces surrounding two or more stacked items indicate conflicting options, one of which *must* be chosen.
FILE (file-reference)	An uppercase word or phrase indicates a keyword that must be entered as shown; a lowercase word or phrase indicates an item for which a variable value must be supplied.

Chapter 1

Introduction to Program Development on VAX/VMS

The VAX-11 operating system, VAX/VMS, and its command language, DCL, provide numerous tools and utilities for program development. This chapter summarizes the basic things you need to know to use the command language in developing and testing your PL/I programs, including

- The commands you use to create, compile, link, and execute PL/I programs.
- The rules for specifying input and output files for commands and programs.
- The commands available to you for file creation, modification, and maintenance.

For a tutorial introduction to these concepts, see the *VAX/VMS Primer*. For detailed definitions of commands and file specifications, see the *VAX/VMS Command Language User's Guide*.

1.1 VAX/VMS Commands for Program Development

Figure 1-1 illustrates the DCL commands you use to create and run PL/I programs. (Section 1.2 explains how each command finds and creates appropriate files.) The commands are shown in their simplest forms. You can also specify qualifiers on the commands to request special processing or to indicate a special type of input file, as in these examples:

```
$ PLI/LIST=LP: METRIC

$ LINK METRIC,MYLIB/LIBRARY
```

In the PLI command example, the /LIST qualifier requests the compiler to create a listing file for the source program METRIC.PLI and to output the file on a line printer (LP: is the device name for line printers). In the LINK command example, the /LIBRARY qualifier indicates that the input file MYLIB is a program library consisting of object modules. When this command is executed, the linker will automatically search this library to locate external procedures and external variables that are referenced in the source file METRIC.PLI.

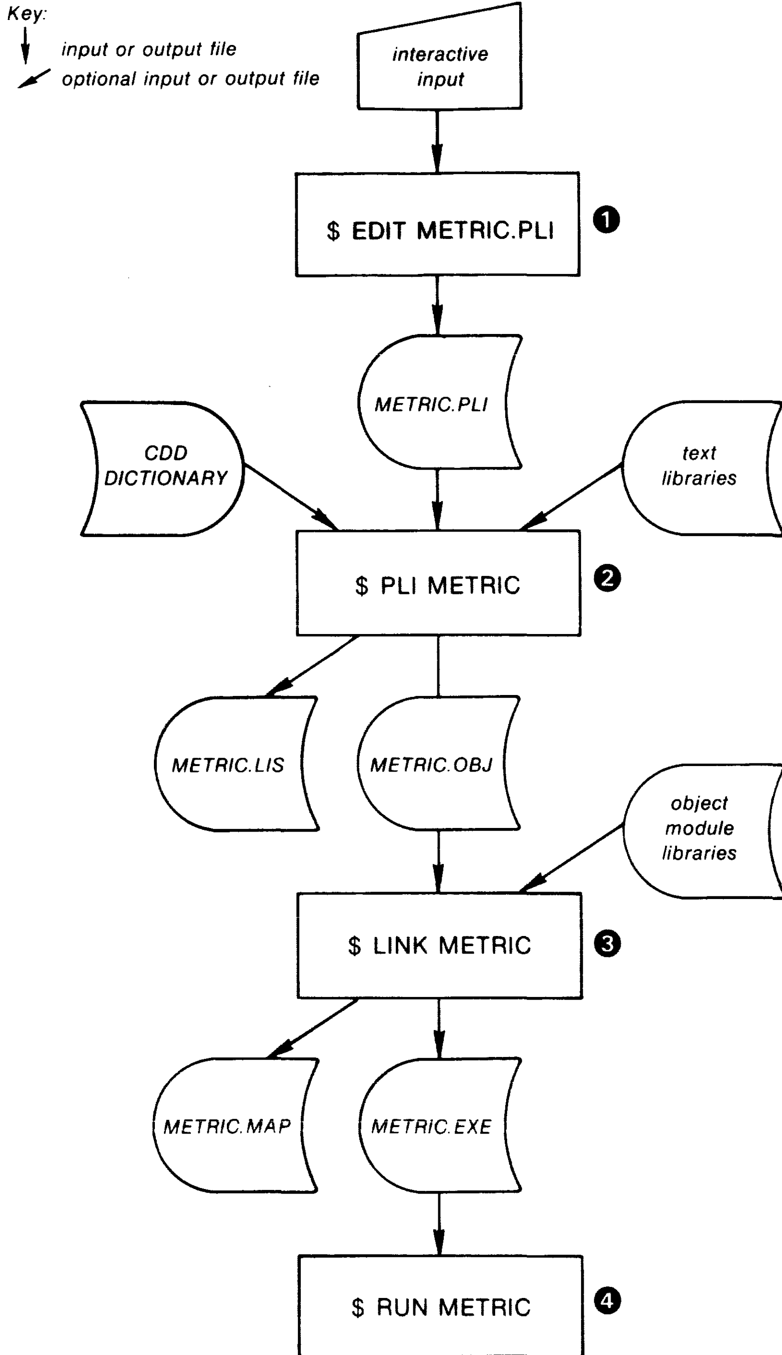


Figure 1-1: Commands for PL/I Program Development

❶ The EDIT command invokes a system editor to create a disk file containing PL/I source statements.

❷ *The PLI command invokes the PL/I compiler to process the source statements and verify that there are no syntax errors or violations of the language rules. It searches user-specified libraries and CDD dictionaries, if any, and default libraries to locate INCLUDE files referenced in the source program. If there are no errors, the compiler creates an object module and optionally a listing.*

❸ The LINK command binds object modules into an executable program image. The linker searches system libraries and user-specified libraries, if any, to locate all run-time modules, external procedures, external variables, and global symbols required for the image.

If an error occurs, you may need to reissue the LINK command, specifying other object modules or libraries that contain needed definitions.

❹ The RUN command executes a program image.

If your program fails or produces unexpected output, it probably has an error. After you determine the cause of the error, you can correct the source program, recompile, and relink the program.

The commands shown in Figure 1–1, and others of specific interest to PL/I programmers, are described in detail in Chapters 2 through 6.

1.1.1 Hints for Entering Commands

Note the following hints for entering commands:

- You can truncate (shorten) any command name or qualifier name to four characters. In some cases, fewer than four characters are accepted, so long as there is no ambiguity about the name.
- You must precede each qualifier name with a single slash character (/).
- If you omit a required parameter, for example, a file specification, the DCL command interpreter will prompt you to enter it.
- You can enter a command on as many lines as you wish, as long as you end each continued line with a hyphen (-). The command interpreter prompts for the rest of the command with the characters $_.
- After you have entered a complete command, you must press RET to pass the command to the system for processing.
- You can cancel a command before the final RET by using CTRL/Y.
- You can interrupt command execution by using CTRL/Y. To resume the interrupted command, enter the CONTINUE command. To stop processing completely after pressing CTRL/Y, you can begin entering other DCL commands.

If you make an error entering a command, for example if you misspell a command or qualifier name, the command interpreter issues an error message and you must reenter the entire command string.

1.1.2 HELP

You can obtain online information about a command, its parameters, and its qualifiers by entering the HELP command. When you request help on a command name, HELP displays a brief description of the command and lists the additional information available. For example, you can enter

```
$ HELP PRINT
```

The HELP command then displays a description of the PRINT command and a list of its qualifiers. To get further information, you must reenter the HELP PRINT command with the name of the qualifier you want information about. For example:

```
$ HELP PRINT /JOB_COUNT
```

The HELP command also provides detailed information about the PLI command and the VAX-11 PL/I language. You can obtain information about PL/I topics by specifying a PL/I keyword. For example:

```
$ HELP PLI STATEMENTS
```

Information is also available at nested levels, for example:

```
$ HELP PLI QUALIFIERS /SHOW
```

This command causes the valid options of the VAX-11 PL/I /SHOW qualifier to be displayed.

1.2 File Specifications and Defaults

A file specification provides VAX/VMS with all the information it needs to locate a unique file. To define a unique PL/I source file, you need only give the file a unique name and a file type of PLI. All other portions of a file specification can default to system- and command-supplied names.

In Figure 1-1, the following DCL commands appear:

```
$ EDIT METRIC.PLI
$ PLI METRIC
$ LINK METRIC
$ RUN METRIC
```

For these commands, defaults are in effect as follows:

- For all the commands shown, the system uses the current default device and directory to locate a file specified.
- The EDIT command does not assume a default file type. Here, and in Figure 1-1, the file type PLI is specified because it is the default file type for the PLI command.
- The PLI command assumes a default input file type of PLI. Unless you use qualifiers on the PLI command to change the output file types, the compiler uses LIS and OBJ for the listing and object files, respectively.
- The LINK command assumes a default input file type of OBJ. Unless overridden by qualifiers, the default file types EXE and MAP are used by the linker for the image and map files, respectively.
- The RUN command assumes a default input file type of EXE. file type is EXE.

Table 1-1 summarizes the syntax of VAX/VMS file specifications, giving a description of each field in a file specification and a summary of the defaults applied. The following subsections provide additional examples of file specifications and explanations of some useful applications of defaults.

Table 1-1: Summary of File Specification Syntax

Field	Syntax Rules	Defaults	Notes
node	1 - 6 characters terminated by ::	local node	node::node:: defines a path node"access-control":: *in VAX/VMS, username password* node::"non-VMS-file-specification"
device dev c u	valid mnemonic or logical name A - Z 0 - 65535	SYS$DISK A 0	CR -card reader NET -network device DB -disk device MB -mailbox DM -RK06/7 disk MT -magnetic tape DX -floppy disk TT -terminal LP -line printer TU -cartridge tape
directory [name] [name.name...]	1 - 9 characters up to 8 names, separated by periods (.)	current default	[*] all directories [name...] all directories in path [*...] all subdirectories in all directories [-.name] back up a directory

Table 1-1 (Cont.): Summary of File Specification Syntax

<table>
<tr><th>Field</th><th>Syntax Rules</th><th>Defaults</th><th>Notes</th></tr>
<tr><td>filename</td><td>0 - 9 characters</td><td>Input: temporary defaults apply
Output: same as input file</td><td>* — all file names
string — match all names containing 'string'
str%ng — match any character in % position</td></tr>
<tr><td>filetype</td><td>0 - 3 characters preceded by .</td><td>Applied by command; temporary defaults apply</td><td>Wildcard rules same as for filename
<table>
<tr><th>Command</th><th>Input</th><th>Output</th></tr>
<tr><td>PLI</td><td>PLI,TLB</td><td>OBJ,LIS</td></tr>
<tr><td>LINK</td><td>OBJ, OLB</td><td>EXE, MAP</td></tr>
<tr><td>LIBRARY</td><td>OBJ</td><td>OLB</td></tr>
<tr><td>LIBRARY/TEXT</td><td>TXT</td><td>TLB</td></tr>
<tr><td>RUN</td><td>EXE</td><td></td></tr>
<tr><td>PRINT, TYPE</td><td>LIS</td><td></td></tr>
</table></td></tr>
<tr><td>version</td><td>0 - 32767 preceded by ; or .</td><td>Input: highest
Output: highest + 1</td><td>* — all versions
; — use most recent version</td></tr>
</table>

1.2.1 Directories and Subdirectories

A directory file lists files on a device that belong to a particular user or account. VAX/VMS uses the information in the directory file to locate the desired file in the directory.

To specify a directory, enclose its name in square brackets:

```
[PROJECT]
```

If you do not include a directory in a file specification, VAX/VMS uses your current default directory. Section 1.2.2 shows how to change this default.

Within a directory, you can create subdirectories to contain related files. You then refer to the subdirectory by concatenating the directory and subdirectory names, separated by a period. For example:

```
[PROJECT.SOURCE]
```

Use the CREATE/DIRECTORY command to create a subdirectory. For example, the commands

```
$ CREATE/DIRECTORY [PROJECT.SOURCE]
$ CREATE/DIRECTORY [PROJECT.OBJECT]
$ CREATE/DIRECTORY [PROJECT.LIST]
```

create the three subdirectories shown, which can then be used to contain source files, object files, and listing files, respectively. Once you have created a subdirectory, you can copy or rename files into it, list the files it contains, make it your default directory, and use it in any other way as you would a main directory.

1.2.2 Changing the Default Directory

To change the default device or directory that VAX/VMS applies to all file specifications, use the SET DEFAULT command. Unless overridden in the explicit specification of an individual file, defaults set by this command remain in effect for all subsequent commands until you either issue a new SET DEFAULT command or log off the system. For example:

```
$ SET DEFAULT [PROJECT.SOURCE]
$ PLI METRIC
```

The PLI command compiles the source program METRIC.PLI from the current default directory [PROJECT.SOURCE]. The output file, METRIC.OBJ, is also placed in this directory.

1.2.3 Logical Names

An alternative way of referring to a specific device, directory, or file is to use a logical name. A logical name can represent an entire file specification

or the leftmost portion of one. You can create logical names with the DEFINE command. For example:

```
$ DEFINE SRC [PROJECT.SOURCE]
$ TYPE SRC:ALPHA.PLI
```

The DEFINE command creates the logical name SRC to represent the directory specification [PROJECT.SOURCE]. When SRC is used in the TYPE command, the system translates it: that is, the logical name in the file specification is replaced by its current equivalence name. The TYPE command displays the file [PROJECT.SOURCE]ALPHA.PLI.

Only one logical name is permitted in a file specification. The name must be the first or only element of the file specification, and it must be followed by a colon if any other elements are present.

The VAX/VMS system maintains tables of all logical names that are created by users. There are three kinds of logical name table:

- Process logical name tables. A separate logical name table exists for every user, or process, on the system. Names in a process logical name table are available only to the user who defines them. A DEFINE command places a logical name in your process logical name table by default.
- Group logical name tables. A separate logical name table exists for every group in the system. The names in any of these tables can be accessed only by users who have the same group number in their user identification code.
- System logical name table. There is only one system logical name table. The logical names in this table can be accessed by all users.

When the system translates a logical name, it first searches the process, then group, then system logical name tables, in that order, for a logical name. Each time the system translates a logical name, it checks to see if the result is itself a logical name. If so, the system translates the result. Therefore, you can define a logical name in terms of another logical name.

You can determine the current equivalence for a logical name by entering the SHOW TRANSLATION command. For example:

```
$ SHOW TRANSLATION SRC
  SRC = [PROJECT.SRC]     (process)
```

VAX/VMS system programs use logical names in many ways. For example, the PL/I compiler and the linker use them to provide default libraries for INCLUDE modules and object module libraries, respectively. (See Sections 3.3 and 4.3 for a full description.)

A principal use for programmers is to provide device and file independence for executable program images or command procedures. For example, the

name you give a file constant in a PL/I source program can be a logical name: each time you execute the program, you can issue a DEFINE command to provide a new equivalence name for the PL/I file. The relationship between PL/I file constants and VAX/VMS file specifications is described in Section 16.2.

Table 1-2 lists the DCL commands for creating, deleting, and examining logical names.

Table 1-2: Commands for Maintaining Logical Names

Command	Function
DEFINE	Creates a logical name and places it in the specified logical name table. The /PROCESS, /GROUP, and /SYSTEM qualifiers specify which table.
DEFINE/USER	Creates a logical name for the execution of the next image only. The name is automatically deleted after the next command or program is executed.
ASSIGN	Performs the same function as DEFINE. However, the order of the command parameters is reversed.
DEASSIGN	Deletes a logical name from the process, group, or system logical name table.
SHOW TRANSLATION	Displays the result of translating a logical name once, and displays the name of the table in which the result was found. This command can be issued when a program is interrupted with CTRL/Y without terminating the program (see Section 5.1.3.).
SHOW LOGICAL	Displays the result of translating a logical name recursively. The SHOW LOGICAL command causes the current image that is executing, if any, to be terminated (see Section 5.1.3).

Sections 16.2.2 and 16.2.3 describe uses of logical names that are of special interest to the PL/I programmer.

1.3 File Creation and Maintenance

Table 1-3 describes some of the basic file-handling commands available to programmers in DCL. For online assistance in entering a command or determining its parameters, qualifiers, or options, use the HELP command at a terminal.

Table 1-3: VAX/VMS Commands for File Maintenance

Category	Command	Command Function
File creation	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 VAX/VMS interactive editing programs, for example, SOS or EDT.
Correcting and modifying files	EDIT[/editor]	Invokes one of the interactive editors to make changes or additions to a disk file.
Cataloging and organizing files	CREATE/DIRECTORY	Establishes a new directory or a 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.

Table 1-3 (Cont.): VAX/VMS Commands for File Maintenance

Category	Command	Command Function
Cataloging and organizing files (Cont.)	LIBRARY	Creates and maintains libraries of INCLUDE text modules and libraries of object modules.
	RENAME	Changes the directory in which a file is cataloged; or changes the file name, file type, or version number of a file or files.
	SET DEFAULT	Changes the current default device or directory.
Copying and backing up files	{ ALLOCATE BACKUP INITIALIZE MOUNT }	Provide device-handling and control commands that let you access data written on nonsystem disks, on magnetic tapes, or on punched cards; or that output data to a disk or tape.
	COPY	Copies the contents of a file or files to another file or files.
Deleting files	DELETE	Makes the contents of a file inaccessible by removing its directory entry.
	PURGE	Deletes a specified number of earlier versions of a file or a group of files.

1.4 Command Procedures

A command procedure is a file that contains a sequence of VAX/VMS commands and, optionally, data. You can cause the commands in the procedure to be executed in either of two ways:

- Interactively: you specify the name of the file following the @ (Execute Procedure) command. For example:

  ```
  $ @TESTAM
  ```

  The @ command assumes that the file type of the specified command procedure is COM. This command executes the procedure TESTAM.COM.

- You can submit the command procedure to a system batch job queue for execution. After the job completes, the system prints a log file that indicates how the job ran. The SUBMIT command submits a job. For example:

  ```
  $ SUBMIT TESTAM
  ```

  This command places the file TESTAM.COM in the system batch job queue.

The following subsections contain two examples of command procedures.

1.4.1 Command Procedures for Program Development

You can devise command procedures to simplify and enhance your program development. For example, you can write a command procedure that will invoke an editor with which you can create a PL/I source file, and, when you exit from the editor, will automatically compile, link, and run your program. The command procedure can specify all the needed libraries for the PLI and LINK commands, and can even contain all the input data required to test the program.

Command procedures can also be generalized. By taking advantage of such DCL commands as the assignment statement and the IF, GOTO, and ON commands, you can write a command procedure that looks like a PL/I program: it can process variables, make decisions based on their values, and handle errors.

The following example will give you an idea of how to construct command procedures to help you with your PL/I program development and testing. The procedure issues all the DCL commands necessary to create and test a single-module, interactive program. The notes following the example are keyed to the numbered lines of the example.

```
$ ON WARNING THEN EXIT        ❶
$ IF P1 .NES. "" THEN GOTO EDIT
$ INQUIRE P1 "File name: "         ❷
$ EDIT:
$ DEFINE/USER SYS$INPUT SYS$COMMAND      ❸
$ EDIT 'P1'.PLI    ❹
$ WRITE SYS$OUTPUT "Beginning compile..."      ❺
$ PLI 'P1'      ❻
$ WRITE SYS$OUTPUT "Beginning link..."
$ LINK 'P1'      ❼
$ DEL 'P1'.OBJ;*      ❽
$ WRITE SYS$OUTPUT "Beginning run..."
$ DEFINE/USER SYS$INPUT SYS$COMMAND      ❾
$ RUN 'P1'      ❿
$ INQUIRE CLEANUP "Purge previous versions?"      ⓫
$ IF CLEANUP THEN PURGE 'P1'.*
```

❶ This command establishes the way the command procedure deals with errors that occur during its execution. Should any command return a severity of WARNING or worse, execution of the command procedure will cease.

Each command that is intended for the command interpreter (all commands, in this example) must begin with a dollar sign ($).

❷ These three commands establish the name of the PL/I source file. You can supply parameters when you invoke a command procedure. These parameters are assigned to symbols named P1, P2, and so on up to P8. For example, if the command procedure above were named P.COM and you wanted to work on a source file named METRIC.PLI, you could issue the command

```
$ @P METRIC
```

This command would invoke P.COM and assign the value METRIC to P1.

If you do not supply a parameter with the command, the symbol P1 is null. The first command of the three tests whether P1 is null. If not, the command procedure skips to the line labeled "EDIT:". If P1 is null, the procedure requests that you supply a file name, assigns the file name to P1, and only then proceeds to EDIT:. This dialog appears as follows on your terminal:

```
$ @P
File name: METRIC
```

❸ This command gives the logical name SYS$INPUT the equivalence name of SYS$COMMAND (that is, your terminal) for the duration of the next image, which is the invocation of the editor that follows. The editor, while active, will receive its input from the terminal. If you omit this command, the editor seeks its input from the command procedure and therefore is not usable interactively.

❹ This command invokes a system editor to edit the file having the name you specified (now assigned to P1) and the type PLI. The apostrophes around P1 request substitution; they are required syntax.

❺ This command types the message "Beginning compile..." on your terminal. It does not execute until you have finished using the editor. Since the commands in a command procedure are normally not echoed on your terminal, such messages are helpful for keeping track of the procedure's progress.

❻ This command compiles the source file having the name you specified and (by default) the type PLI. If you customarily use extra qualifiers, libraries, and so on, you can include them here.

❼ This command links the object file having the name you specified and (by default) the type OBJ.

❽ This command deletes the object file, which is no longer necessary after the link operation.

❾ This command serves the same purpose as the one preceding the EDIT command. It equates the default PL/I device SYSIN to your terminal instead of the command file, thus allowing you to enter data for your program from your terminal.

Alternatively, you can include test data for your program in the command procedure. Such data would consist of lines with no preceding dollar signs following the RUN command in the procedure. However, in this case you must omit the DEFINE/USER command.

❿ The RUN command executes the file having the name you specified and (by default) the type EXE.

⓫ These two lines ask you if you wish to purge previous versions of the source and image file. If you answer YES or Y, the PURGE command purges them.

1.4.2 The Login Command Procedure File

When you log in, the system searches for a file in your directory named LOGIN.COM. If such a file exists, the system executes the commands contained in it before giving you control. The login command file is therefore useful for establishing logical names and symbols that you use often.

The login command procedure shown in the following example contains commands that might be of special interest to a programmer. The notes following the example are keyed to the numbered lines.

```
$ Q :== "SHOW QUEUE/BATCH/DEVICES/ALL/FULL"      ❶
$ P :== "@P"      ❷
$ EDT :== "EDIT/EDT"                          }
$ PED :== "EDIT/EDT/COMMAND=PROG.EDT"         } ❸
$ LIST :== "PLI/NOOBJ/LIST=LP:"
$ DEFINE CODE DB1:[PROJECT.SOURCE.PLI]        }
$ DEFINE LISTS DB1:[PROJECT.LISTINGS]         } ❹
$ DEFINE PROG DB1:[PROJECT.IMAGES]            }
$ DEFINE LIB DB1:[PROJECT.LIBRARY]            }
$ DEFINE PLI$LIBRARY LIB:INCFILES.TLB         } ❺
$ DEFINE LNK$LIBRARY LIB:MATHMODS.OLB         }
```

❶ This command and the four that follow it define symbols. Once defined, symbols can replace their equivalent DCL command lines. They provide a convenient shorthand for lengthy, frequently used command lines. The command shown equates the symbol Q to the qualified SHOW QUEUE command, which displays the status of print and batch queues.

❷ This command equates the symbol P to the command @P. When you type P in response to the DCL prompt, DCL executes the command file P.COM from your current directory. (This could be the program development command file shown in Section 1.4.1.) Defining this symbol saves you the trouble of typing the at-sign (@).

❸ These three commands equate symbols to commonly used command lines. The first symbol invokes EDT, a system editor. The second symbol invokes EDT with a special startup command file. The third symbol invokes the PL/I compiler to compile a source file and produce a listing on the line printer without creating an object or listing file. When you use these symbols, you type a file specification following the symbol. To use LIST, you would type

```
$ LIST METRIC
```

❹ These four commands define four logical names. Once they are defined, you can use them at the beginning of file specifications to save yourself the trouble of typing all the device and directory information. The command

```
$ PLI CODE:METRIC
```

would be equivalent to

```
$ PLI DB1:[PROJECT.SOURCE.PLI]METRIC
```

❺ These two commands define the logical names PLI$LIBRARY and LNK$LIBRARY. Note that they use the logical name LIB, defined in the previous line. These two logical names are default library specifications for the PL/I compiler and the linker, respectively. The PL/I compiler searches PLI$LIBRARY to locate INCLUDE modules that it cannot find by searching text libraries specified in the PLI command. The linker searches LNK$LIBRARY to resolve references that it cannot resolve by searching libraries and modules specified in the LINK command.

Chapter 2

Creating and Correcting Programs

The first step in developing a VAX-11 PL/I program consists of creating the program's source file. VAX/VMS offers two supported text editors that allow you to do this: SOS and EDT. This chapter provides an introduction to the use of EDT. For information on SOS, refer to the *VAX-11 SOS Text Editing Reference Manual.*

There are three other sources of information on EDT available to you. The first is the *VAX-11 EDT Editor Reference Manual.*[1] The second is the computer-assisted course titled "Introduction to the EDT Editor" supplied with the VAX/VMS operating system. The third is EDT's help facility, described in Section 2.1.2.

2.1 Introduction to EDT

EDT, the DEC Standard Editor, is an interactive general-purpose text editor. It offers two modes of operation: line editing, in which operations are performed on entire lines of text; and character editing, in which operations are performed on characters and words as well as on lines. Line editing is possible on either hardcopy or video terminals. Character editing, while usable on hardcopy terminals, is most effective on video terminals.

Line editing mode, with its English-like commands, is simple for the inexperienced user to learn. Character editing mode, while requiring practice, is also very simple. Therefore, EDT is a good editor for someone who must learn a text editor quickly.

---

1. Some installations may have the *EDT Editor Manual* instead of the *VAX-11 EDT Editor Reference Manual.* The two manuals contain the same information about EDT.

EDT also offers many advanced features for more experienced users:

- Multiple text buffers. By default, editing operations take place within a single text buffer called MAIN. However, you can maintain an unlimited number of alternate text buffers as "holding areas" for text that you do not necessarily wish to incorporate in the output file.
- Flexible input and output commands. You can copy files into an EDT text buffer after beginning the editing session, and you can output text buffers (or portions of text buffers) to files before ending the session.
- Macro capability. You can create sequences of line editing commands that you invoke with a single command.
- The ability to define keys for custom character editing applications. For example, a keypad key can be defined so that it inserts a specified line of text each time it is pressed. This function is especially useful in programming applications where certain statements may be repeated frequently.

Finally, EDT protects your text. Should your editing session end in an unexpected manner, you can recover all your editing operations by reentering the EDT command line with the /RECOVER qualifier. EDT then "replays" your editing session up to the point of interruption, using the contents of the journal file that it maintained during the lost session.

The following subsections introduce EDT's line editing commands and help facilities.

2.1.1 Line Editing Command Summary

When you invoke EDT, and throughout your editing session, EDT prompts you to enter line editing commands by displaying an asterisk. For example:

```
$ EDIT/EDT METRIC.PLI
    1           METRIC: PROCEDURE OPTIONS(MAIN);
*
```

Table 2-1 describes briefly (in alphabetical order) the most useful commands that you can enter in response to the line editing prompt (*). Examples of these commands occur throughout Sections 2.2, 2.3 and 2.4. Each command has a smallest acceptable abbreviation, shown in bold type in the table.

All line editing commands are terminated with a RET. Most of the commands allow or require you to specify a range or ranges; the range specification tells EDT where the action of the command should take place. Section 2.4.1 summarizes range specifications, and the command examples show various ways of specifying a range.

Table 2-1: Summary of Line Editing Commands

Command	Function
CHANGE [range]	Invokes character editing mode for specified buffer
CLEAR	Deletes the contents of a text buffer
COPY [range1] **TO** [range2] [/**Q**UERY]	Copies lines specified by range1 to a location in an EDT buffer specified by range2; does not delete lines from original location
DEFINE { KEY / MACRO }	Defines a new or revised key function for character editing mode, or defines a macro name
DELETE [range] [/**Q**UERY]	Deletes a specified line or lines
EXIT [file-spec]	Terminates EDT, saving the contents of the text buffer MAIN as the output file
FILL [range]	Reformats a block of text so a maximum number of full words fill a line as without exceeding the right margin
FIND range	Moves the current line to a specified line
HELP [topic ...]	Displays information on the specified EDT command or function
INCLUDE file-spec [range]	Copies an external file to a location in a text buffer specified by range
INSERT [range]	Opens a text buffer for the insertion of text at the location specified by range
MOVE [range1] **TO** [range2] [/**Q**UERY]	Moves lines specified by range1 to the location specified by range2, deleting the lines from the source location
PRINT file-spec [range]	Creates a listing file with the specified file name
QUIT [/SAVE]	Terminates EDT without creating an output file, optionally saving the journal file
REPLACE [range]	Deletes specified lines from a text buffer and leaves the buffer open for insertion of text
RESEQUENCE [range]	Assigns new line numbers to a range of lines
SET [parameter]	Sets a variety of editor operating parameters
SET [NO]NUMBER]	Enables/disables the display of line numbers
SHOW [parameter]	Displays specified editor operating parameters
SUBSTITUTE /string1/string2/[range] [/**Q**UERY]	Replaces string1 with string2, either in the current line or in the specified range
[SUBSTITUTE] NEXT [/string1/string2]	Replaces string1 with string2, based either on the strings specified or on the previous SUBSTITUTE command

Table 2-1 (Cont.): **Summary of Line Editing Commands**

Command	Function
TAB·ADJUST [-]n [range]	Shifts each of a range of lines a specified number of logical tab stops
[TYPE] [range]	Displays specified lines and makes the first line in range the current line; the default command
WRITE file-spec [range]	Moves a copy of specified text from a buffer to a file

2.1.2 The Help Facilities

EDT offers online help in both line and character editing modes. In line editing mode, you invoke the help facility by entering the HELP command. Issued without parameters, this command displays information on how to get further help, plus a list of subjects for which help is available. If you enter one of the subjects as a parameter to the HELP command, EDT displays information on that subject, and possibly another list. For example:

```
*HELP DELETE

DELETE

The DELETE (abbreviation: D) command deletes the line specified
    .
    .
    .
Additional information available:

/QUERY
*HELP DELETE /QUERY

DELETE

/QUERY
    .
    .
    .
         Q    Quit, do not delete any of the rest of the
              lines
         A    All, delete all the rest of the lines
*
```

In character editing mode, you obtain help by pressing the HELP key on your keypad; EDT will display a diagram of the keypad with all the key functions identified. You can then obtain help on an individual function by pressing the key that invokes that function. (Section 2.5 shows you how to find the HELP key.)

2.2 Invoking and Terminating EDT

An editing session begins when you invoke EDT with the EDIT/EDT command, and ends when you terminate EDT with the EXIT or QUIT command. You may start an editing session with no file and create the text for the file during the course of the session. Or you may specify an existing file when you start the session, in which case EDT loads the file into its MAIN text buffer. EDT does not destroy the contents of any existing file that you edit; it simply produces a new version, leaving the old version intact.

2.2.1 Invoking EDT

To invoke EDT, issue an EDIT/EDT command in the format

```
EDIT/EDT[/qualifier...] file-spec
```

Qualifiers	Defaults
/[NO]COMMAND[=file-spec]	/COMMAND=EDTINI.EDT
/[NO]JOURNAL[=file-spec]	/JOURNAL=infile-name.JOU
/[NO]OUTPUT[=file-spec]	/OUTPUT=infile-spec
/[NO]READ_ONLY	/NOREAD_ONLY
/[NO]RECOVER	/NORECOVER

file-spec

Specifies the file to be created or edited. If the file does not exist, EDT creates it.

EDT does not provide a default file type. If you do not specify one, the file type is null.

/OUTPUT[=file-spec]
/NOOUTPUT

Supplies an alternate file specification for the output file. By default, EDT creates an output file upon exit that has the same name and type as the input file and a version number of 1 (if the input file does not exist) or one higher than the highest existing version (if the input file does exist).

If you specify /NOOUTPUT, EDT does not automatically create an output file when you issue the EXIT command.

The remaining qualifiers, which describe specialized editor functions, are described elsewhere: the /COMMAND qualifier, in Section 2.7.3; the /JOURNAL, /READ_ONLY, and /RECOVER qualifiers, in Section 2.6.

For convenience, you can issue the following command to equate a short command symbol (EDT, in this example) to EDIT/EDT:

```
$ EDT :== "EDIT/EDT"
```

After you issue this command, the command interpreter will recognize the symbol EDT (or any other symbol you specify) as equivalent to EDIT/EDT.

When you invoke EDT, the response varies depending on whether or not the file that you specify exists. (Other factors, such as commands contained in a startup command file named EDTINI.EDT, may further alter the response.) If the file does not exist, EDT so informs you, and prompts you to issue editing commands:

```
$ EDIT/EDT METRIC.PLI
Input file does not exist
[EOB]
*
```

The asterisk (*) is EDT's line editing prompt. When EDT is displaying the asterisk prompt, you can enter any of the commands listed in Table 2-1.

If the file exists, its first line is displayed instead of [EOB]:

```
$ EDIT/EDT METRIC.PLI
    1           METRIC: PROCEDURE OPTIONS(MAIN);
*
```

NOTE

If you invoke EDT and it does not display an asterisk prompt, you cannot enter line editing commands. This condition can result when the current default directory contains a startup command file named EDTINI.EDT that causes EDT to enter character editing mode directly. If this happens, you can enter line editing mode by typing a CTRL/Z. You can override the unwanted effects of a startup command file by including the /NOCOMMAND qualifier on the command line.

2.2.2 Terminating EDT

Use the EXIT command to terminate EDT and create an output file from the contents of the MAIN text buffer. To override the default output file, you can specify an output file with the EXIT command, as shown in the following example:

```
*EXIT ALTNAME.PLI
_DB1:[PROJECT]ALTNAME.PLI;1    55 lines
$
```

The QUIT command terminates EDT without creating an output file. You can use QUIT if you are simply reading a file without modifying it or if you do not want to save your edits.

2.3 Creating a New File in Line Mode

To create a new file, you issue an EDIT/EDT command that specifies a file that does not currently exist in your directory. After EDT responds with the asterisk prompt, issue the INSERT command (abbreviation I) followed by RET. The cursor or print head then moves to the right 16 spaces; this space is left by EDT to accommodate line numbers, although none appear at this stage. You can now enter as many lines of text as you wish. When you are finished entering text, terminate the insert with CTRL/Z. The following example illustrates this process:

```
$ EDIT/EDT EXAMPLE.TXT
Input file does not exist
[EOB]
*I
                This is the first line of EXAMPLE.TXT
                This is the second line of EXAMPLE.TXT
                This is the third line of EXAMPLE.TXT
                This is the fourth line of EXAMPLE.TXT
                This is the fifth line of EXAMPLE.TXT
                This is the sixth line of EXAMPLE.TXT
                This is the seventh line of EXAMPLE.TXT
                CTRL/Z ^Z
*
```

The [EOB] designation indicates that you are currently at end-of-buffer, and that any text you insert will be the only text in the buffer.

If you do not want EDT to leave space in front of each line for line numbers, you can issue the SET NONUMBERS command; EDT will then begin each line at the left margin of the terminal. EDT continues to number lines, but does not display the numbers. You can restore the line number display later by issuing a SET NUMBERS command.

2.4 Editing an Existing File in Line Mode

To edit an existing file in your directory, issue an EDIT/EDT command that specifies its name. (To edit a file from a directory other than your own, see Section 2.4.8.) EDT displays the first line in the file, as shown in the following example:

```
$ EDIT/EDT EXAMPLE.TXT
    1           This is the first line of EXAMPLE.TXT
*
```

The number 1 to the left of the line is the line number. It is not part of the file. The file starts with the word *This*.

The line displayed is the current line. EDT uses the current line as the default in many of its operations. For example, an INSERT command that does not specify a range causes EDT to insert text in front of the current line.

The concept of "range" is central to all EDT line editing operations. The next section describes ways of specifying range. The sections that follow describe the most common and useful line editing operations.

2.4.1 Range Specifications

A range is the line or lines on which EDT performs an operation. A range specification is a description of a range in terms that EDT can understand. All the line editing commands (except SUBSTITUTE NEXT) described in the sections that follow accept one or more range specifications, although many do not require one.

The simplest range specification identifies a single line of text. A line can be located by its position in the file relative to the current line, by a text string that it must contain, and by its line number. Since line numbers are primarily useful in range specifications, they are described here.

When you insert lines of text in a new file, or when EDT loads an existing file into its MAIN buffer, each line of the file receives a number. The numbering starts with 1 and is incremented by ones. If you insert lines of text between existing lines, EDT numbers the new lines using appropriate decimal increments. This technique ensures that there will be enough unique line numbers to cover any reasonable editing operation. EDT displays the line numbers whenever it displays text, unless you have issued the SET NONUMBERS command. In that case, EDT does not display line numbers, but it does continue to assign them.

Single-line range specifications are listed in Table 2-2; examples appear below.

Table 2-2: Single-Line Range Specifications

Specification	Meaning
.	The current line
number	The line specified by the number
'string' or "string"	The next line containing the string you specify
-'string' or -"string"	The preceding line containing the string you specify

Table 2-2 (Cont.): Single-Line Range Specifications

Specification	Meaning
[range] { + / - } [number]	The line that is the specified number of lines after (or before, if minus) the single line specified by range (range defaults to the current line; number defaults to 1)
BEGIN	The first line in the text buffer
END	An empty line (designated by [EOB]) following the last line of text in the text buffer

Specification	**Meaning**
20.6	The line numbered 20.6
'INSERT:'	The next line that contains the string *INSERT:*
"-GET LIST (A);"	The first preceding line that contains the string *GET LIST (A);*
-6	The line six lines before the current line
'PUSH:'+4	The line four lines after the line that contains the string *PUSH:*

When EDT searches for a string, the case of the search string need not match the case of the target. For example, *get list* is a match for *GET LIST* or *Get List*. This condition is the default; you can change it with the SET SEARCH command.

There are several methods available for specifying a range of more than one line. They are listed in Table 2-3; examples appear below.

Table 2-3: Multiple-Line Range Specifications

Specification	Meaning
[range1] { : / , / THRU } [range2]	The set of lines from range1 through range2, which are single line range specifications (both range1 and range2 default to the current line, if omitted)
[range] { # / FOR } number	The specified number of lines beginning with the single line specified by range (range defaults to the current line, if omitted)
BEFORE	All lines in the buffer that precede the current line
REST	The current line and all lines in the buffer that follow it

Table 2-3 (Cont.): Multiple-Line Range Specifications

Specification	Meaning
WHOLE	The entire buffer
range, range... or range AND range AND...	All lines specified by each single line range
[range] ALL 'string']	All lines in the range containing the specified string (the default for range is the entire buffer)

Specification	Meaning
2:6.5	Lines 2 through 6.5, inclusive
'INSERT:'#5	The line containing the string *INSERT:* and the four lines following it, for a total of five lines
.-10:.	The line 10 lines before the current line through the current line, inclusive
10:50 ALL 'GET'	All lines from line 10 through line 50 that contain the string *GET*

Most range specifications can be combined with a text buffer specification. During your editing session, you may wish to hold and edit text in buffers other than MAIN. To create and gain access to alternate buffers, include the name of the buffer in a range specification, using the following syntax:

```
=buffer [range]
 or
BUFFER buffer [range]
```

In this syntax, "buffer" stands for the name of the buffer. It can be from 1 to 30 alphanumeric characters, but it must start with an alphabetic character. If you include a range of lines following the buffer name, you specify the range within the named buffer. If you omit the range specification, you specify either the entire named buffer or its first line, depending on context.

The following examples show buffer specifications in use.

Specification	Meaning
=PROG1	The entire contents of the text buffer named PROG1, or (for commands requiring a single-line range specification) its first line
=INC'SUB1:':'RETURN'	The lines that contain the strings *SUB1* and *RETURN* in the text buffer named INC, and all lines between
=COM ALL 'COPY'	All lines that contain the string *COPY* in the buffer named COM

2.4.2 Maneuvering in the File

This section describes commands for maneuvering in a buffer containing text, in other words, for changing the location of the current line.

The TYPE command, followed by a range, causes EDT to display the line or lines in the range and resets the current line to the first (or only) line displayed. The word TYPE (abbreviation T) is optional: it need not be entered. For example:

```
*T 1:3
    1           This is the first line of EXAMPLE.TXT
    2           This is the second line of EXAMPLE.TXT
    3           This is the third line of EXAMPLE.TXT
*4#2
    4           This is the fourth line of EXAMPLE.TXT
    5           This is the fifth line of EXAMPLE.TXT
*
```

If you do not include the word TYPE, and if the range specification begins with an alphabetic character (such as WHOLE or REST), you must precede it with a percent sign (%). Otherwise, EDT tries to interpret the range specification as a command. For example:

```
*REST

Unrecognized command
*%REST
    4           This is the fourth line of EXAMPLE.TXT
    5           This is the fifth line of EXAMPLE.TXT
    6           This is the sixth line of EXAMPLE.TXT
    7           This is the seventh line of EXAMPLE.TXT
*
```

A carriage return in response to the asterisk prompt displays the line following the current line and sets the current line to the displayed line. A series of carriage returns, therefore, displays successive lines and sets the current line to the displayed line each time. This is an easy way to work through a file line by line. For example:

```
*(RET)
    5           This is the fifth line of EXAMPLE.TXT
*(RET)
    6           This is the sixth line of EXAMPLE.TXT
*
```

The FIND command (abbreviation F) locates a specified line without displaying it. It is useful for setting the current line to the top of a large block of text that would be cumbersome to display on the terminal. For example, each of the following commands resets the current line to the top of the MAIN text buffer:

```
*=MAIN

*F =MAIN
```

However, the first command (an implied TYPE command) displays the entire contents of the MAIN text buffer. The second command just sets the current line and displays an asterisk prompt.

If you specify a range that EDT cannot locate, EDT issues a message and does not change the current line setting.

2.4.3 Inserting New Text

The procedure for inserting new text in a buffer already containing text is exactly the same as that for inserting text in an empty buffer (see Section 2.3), except that you can control where the text goes by including a range specification with the INSERT command. The lines you insert are placed in front of the line you specify. If you specify multiple lines, the insert goes in front of the first line in the range. If you omit the range specification, the insert goes in front of the current line.

In the following example, the INSERT command causes EDT to insert text in front of line 5 in the current buffer. Then the range specification (an implied TYPE command) causes EDT to display lines 4 through 6, showing the result of the insertion.

```
*I 5
                First insert line
                Second insert line
                Third insert line
                CTRL/Z ^Z
* 4:6
   4            This is the fourth line of EXAMPLE.TXT
   4.1          First insert line
   4.2          Second insert line
   4.3          Third insert line
   5            This is the fifth line of EXAMPLE.TXT
   6            This is the sixth line of EXAMPLE.TXT
*
```

NOTE

EDT, which inserts text in front of the current line, is different from many other text editors that insert text following the current line.

2.4.4 Deleting and Replacing Text

Use the DELETE command (abbreviation D) to delete a specified range. If you omit the range, the DELETE command deletes the current line. After a delete operation, EDT displays the line following the last line deleted; this is the new current line. For example:

```
*D 4.1#2
2 lines deleted
   4.3          Third insert line
*D
1 line deleted
   5            This is the fifth line of EXAMPLE.TXT
*
```

The /QUERY qualifier to the DELETE command causes EDT to prompt you before deleting each line of the range. The prompt is a question mark (?). You can respond to the prompt in one of four ways:

Y (yes) Delete this line
N (no) Do not delete this line
A (all) Delete all remaining lines in the specified range
Q (quit) Quit the delete operation

The REPLACE command (abbreviation R) deletes a specified range and allows you to insert lines to replace those deleted. You terminate the insertion with a CTRL/Z, just as with the INSERT command.

2.4.5 Moving Text

The COPY and MOVE commands (abbreviations CO and M, respectively) allow you to move one or more lines of text from one place in the buffer to another, or from one buffer to another. The effect of these commands is similar; the only difference is that the COPY command does not delete the text from its original location, whereas the MOVE command does.

The following example illustrates both commands, as well as alternative ways of specifying a range:

```
*%WHOLE
    1           This is the first line of EXAMPLE.TXT
    2           This is the second line of EXAMPLE.TXT
    3           This is the third line of EXAMPLE.TXT
    4           This is the fourth line of EXAMPLE.TXT
    5           This is the fifth line of EXAMPLE.TXT
    6           This is the sixth line of EXAMPLE.TXT
    7           This is the seventh line of EXAMPLE.TXT
*COPY 1:3 TO 'SIXTH'
3 lines copied
*5:6
    5           This is the fifth line of EXAMPLE.TXT
    5.1         This is the first line of EXAMPLE.TXT
    5.2         This is the second line of EXAMPLE.TXT
    5.3         This is the third line of EXAMPLE.TXT
    6           This is the sixth line of EXAMPLE.TXT
*M 5.1#3 TO BEGIN
3 lines moved
*%WH
    0.1         This is the first line of EXAMPLE.TXT
    0.2         This is the second line of EXAMPLE.TXT
    0.3         This is the third line of EXAMPLE.TXT
    1           This is the first line of EXAMPLE.TXT
    2           This is the second line of EXAMPLE.TXT
    3           This is the third line of EXAMPLE.TXT
    4           This is the fourth line of EXAMPLE.TXT
    5           This is the fifth line of EXAMPLE.TXT
    6           This is the sixth line of EXAMPLE.TXT
    7           This is the seventh line of EXAMPLE.TXT
*
```

The /QUERY qualifier to either COPY or MOVE causes EDT to prompt you before copying or moving each line of the range. It operates the same way as the /QUERY qualifier to DELETE (see Section 2.4.4).

2.4.6 Substituting Text

Two commands, SUBSTITUTE and SUBSTITUTE NEXT, substitute one string for another within a line or lines. These are the only line editing commands that can alter text within a line, as opposed to changing the

entire line. The SUBSTITUTE command (abbreviation S) operates on the current line or on a specified range; the SUBSTITUTE NEXT command (abbreviation N) makes a substitution at the next opportunity within the buffer.

The format of the SUBSTITUTE command is

```
SUBSTITUTE /string1/string2/[range] [/QUERY]
```

The command finds string1 and substitutes string2 for it. If you do not specify a range, the substitution takes place in the current line. If you do, the command makes every substitution within the range. The following example illustrates the command first without and then with a range specified:

```
*1
    1           This is the first line of EXAMPLE.TXT
*S /first/1st/
    1           This is the 1st line of EXAMPLE.TXT
1 substitution
*S /of/in/4:6
    4           This is the fourth line in EXAMPLE.TXT
    5           This is the fifth line in EXAMPLE.TXT
    6           This is the sixth line in EXAMPLE.TXT
3 substitutions
*
```

Slashes (/) are not the only characters you can use to delimit string1 and string2. Any nonalphanumeric character will work, as long as the delimiters are matched and do not occur in either string. For example, the following command substitutes the string *A/3* for *A/2* in the current line, using dollar signs ($) as delimiters:

```
*S $A/2$A/3$
   25              SIZE = A/3;
1 substitution
*
```

The /QUERY qualifier to SUBSTITUTE causes EDT to prompt you before making each substitution. It operates the same way as the /QUERY qualifier to DELETE (see Section 2.4.4).

The SUBSTITUTE NEXT command (abbreviation N) substitutes for the next occurrence of string1 that it finds in the buffer. If you specify neither string1 nor string2, the command takes the values of both strings from the last SUBSTITUTE command you issued. For example:

```
*N / in/ of/
    4           This is the fourth line of EXAMPLE.TXT
*N
    5           This is the fifth line of EXAMPLE.TXT
*
```

2.4.7 Input from and Output to Files

Two EDT commands, INCLUDE and WRITE, allow you to incorporate text from files and output text to files during your editing session. The INCLUDE command (abbreviation INC) incorporates the contents of a file at a specified location in a text buffer. If you do not want the entire file incorporated in the MAIN text buffer, you can specify an alternate buffer as the range, and then copy the desired portions of the file to their proper places in MAIN. For example:

```
*INC SBRTNES.PLI =SUBS
*
```

This command creates a buffer called SUBS and fills it with the contents of the file SBRTNES.PLI from the EDT default directory (that is, the directory of the input file given with the EDIT/EDT command).

The WRITE command (abbreviation WR) creates a file by copying the contents of a specified range in a text buffer. The text is not deleted from the text buffer and EDT does not terminate following the operation. If you do not specify a range with the command, EDT outputs the entire contents of the current text buffer. The following example shows the command used with a range:

```
*WR ROUTINE1.PLI =SUBS 'ADD:':'RETURN
_DB1:[PROJECT]ROUTINE1.PLI;1 45 lines
*
```

This command creates the file ROUTINE1.PLI from the lines that contain the strings *ADD:* and *RETURN* in the buffer named SUBS, and all lines in between.

Unless you include a directory in the file specification, WRITE always creates the file in your current default directory. This is true even if the input and output files are in another directory.

2.4.8 Editing a File from Another Directory

You can edit a file that exists in another directory and use the /OUTPUT qualifier to EDIT/EDT to direct the output file to your directory. However, EDT uses the directory of the input file that you specify in the EDIT/EDT command line as its default directory. This default has the following effects:

- EDT attempts to create its journal file in its default directory, that is, the other directory. If you do not have the privilege to do this, EDT issues an error message and terminates. You should instead use the /JOURNAL qualifier to place the journal file in your directory. (See Section 2.6 for a description of the journal file and /JOURNAL.)

- If you issue an INCLUDE command and do not specify a directory, EDT attempts to locate the file in its default directory, that is, the other directory. To specify a file in your own directory, use a directory specification with INCLUDE.

In the following example, a user with the account [WILBUR] edits a file from the account [PROJECT]:

```
$ EDIT/EDT [PROJECT]DATADEF.PLI -
$_/OUTPUT=[WILBUR] /JOURNAL=[WILBUR]
    .
    .
    .
*INCLUDE [WILBUR]ENTRIES.PLI
```

The input file for this editing session is [PROJECT]DATADEF.PLI; the output file is [WILBUR]DATADEF.PLI. The INCLUDE command incorporates a file from directory [WILBUR]. If the INCLUDE command had not specified a directory, EDT would have looked for the file [PROJECT]ENTRIES.PLI.

2.5 Character Editing

EDT's character editing mode allows you to perform editing operations at any position in your text instead of line by line. For most applications, especially those requiring extensive detail modification of existing text, character editing is faster and more straightforward than line editing. When you use character editing mode on a video terminal, your screen always contains an accurate picture of the area of the file in which you are working. The terminal's cursor shows exactly where you are at all times.

There are two types of character editing: nokeypad and keypad. Nokeypad character editing works on all terminals, including hardcopy terminals. It requires you to enter short commands through the keyboard and terminate each command with a RET. Keypad character editing works on the VT52 and VT100 video terminals and on terminals that are compatible with them. In keypad editing, you request editor functions by pressing keys on the auxiliary keypad; no RET is required to terminate the command. Anything you type on the keyboard, including carriage returns, is inserted into the file as text.

This section describes only keypad character editing. To learn about nokeypad character editing, read the *VAX-11 EDT Editor Reference Manual*.

The keypads for the VT52 and VT100 (and compatible) terminals are different. Therefore, the following description refers to functions rather than to specific keys. It is a good idea to keep a copy of the appropriate

keypad diagram handy while you are learning character editing. Figures 2-1 and 2-2 contain the keypad diagrams for the VT52 and VT100, respectively. The numbers or characters shown in the upper right of each key correspond to what you see on the key.

Note that most keys perform two functions. To use the upper of the two functions listed, press the key. To use the lower function, first press and release the GOLD key.

2.5.1 Entering and Exiting Character Editing Mode

To enter character editing mode from line editing mode, use the CHANGE command (abbreviation C). When you issue the CHANGE command, the screen first goes blank and then fills with text. You will find the cursor somewhere on the screen, positioned at the current line or the line you specified with the CHANGE command. (If the buffer is empty, the cursor and [EOB] appear at the top of the screen.)

EDT does not display line numbers while in character editing mode, although it does continue to assign them as you insert text.

When you have finished your character editing operations and wish to return to line mode, enter a CTRL/Z. It terminates character editing and causes EDT to display the asterisk prompt. You can then perform line editing operations or end the editing session, as appropriate.

The sections that follow describe some of the character editing operations available to you.

2.5.2 Maneuvering the Cursor

Before performing most character editing operations, you must move the cursor to the location in the file where you wish the operation to take place. There are many ways to move the cursor; experience eventually teaches which is best in a given situation.

The LEFT and RIGHT functions move the cursor one character to the left or right. If the cursor is at the end of a line, the RIGHT function moves it to the beginning of the next line. Conversely, if the cursor is at the beginning of a line, the LEFT function moves it to the end of the previous line.

The UP and DOWN functions move the cursor one line up or down. The column position of the cursor does not change, unless there is no text in the corresponding column above or below. In that case, the cursor moves to the end of the preceding or following line.

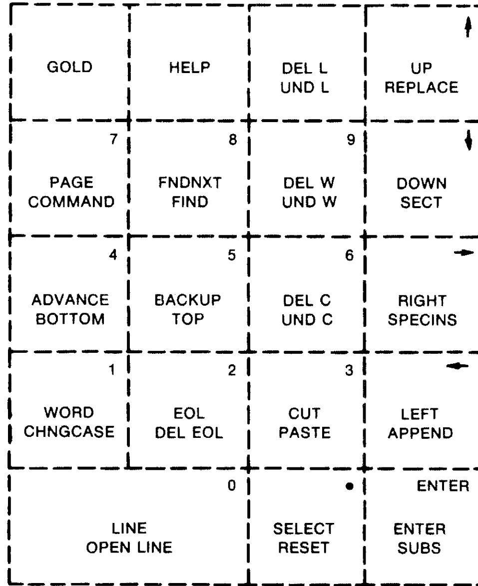
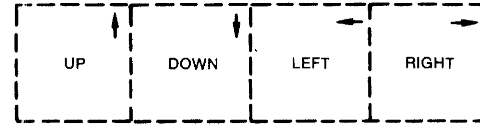


Figure 2-1: VT52 Keypad



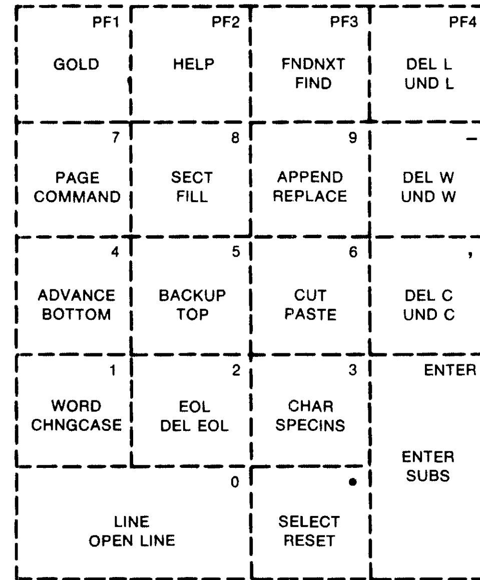


Figure 2-2: VT100 Keypad

The beginning-of-line function, obtained by pressing the BACK SPACE key, moves the cursor to the beginning of the line in which it is positioned. If the cursor is already at the beginning of a line, the function moves it to the beginning of the previous line.

The TOP and BOTTOM functions move the cursor to the beginning and end of the buffer, respectively.

All the remaining cursor movement functions depend in part on the ADVANCE and BACKUP functions. The ADVANCE function causes subsequent cursor movement to occur in the forward direction, that is, toward the end of the buffer. The BACKUP function causes subsequent cursor movement to occur in the backward direction, toward the beginning of the buffer. When character editing begins, cursor movement is forward, until reversed by the BACKUP function.

The following functions depend on the current direction established by ADVANCE and BACKUP:

- The CHAR function moves the cursor one character.
- The WORD function moves the cursor to the beginning of the next or previous word (the end-of-line character is considered a word).
- The LINE function moves the cursor to the beginning of the next line, if the current direction is forward. If backward, the LINE function moves the cursor to the beginning of the line in which the cursor is positioned, or, if the cursor is at the beginning of a line, to the beginning of the previous line.
- The EOL (for end-of-line) function moves the cursor to the next or previous end-of-line character.
- The SECT (for section) function moves the cursor one 16-line section.
- The PAGE function moves the cursor to the next or previous page mark (by default, a form feed).

All of these cursor movement functions can be combined with a repeat count, which causes the function to be repeated a specified number of times. To enter a repeat count, press the GOLD key, then type in the count on the keyboard (not keypad) number keys, then type in the function to be repeated. As you enter the repeat count, the numbers appear on the screen below the area reserved for text. The numbers disappear as soon as you enter the function.

You can also use FIND and FNDNXT (for find next) to move the cursor to a certain string. To find a string, enter the FIND function. EDT prompts you for a search string. Type the search string without delimiters, and terminate it with either the ADVANCE or BACKUP function to determine the direction of search. EDT moves the cursor to the beginning of the

search string. If the search string is not found, EDT issues a message and does not move the cursor.

The FNDNXT function finds the next occurrence of the current search string in the current direction. The current search string is the last string you entered with the FIND function.

Note that you can locate strings that include carriage returns with the FIND function. Simply enter the carriage return as part of the search string. The carriage return does not terminate the search string; you do that with the ADVANCE or BACKUP function. EDT echoes a carriage return in a search string as ^M.

2.5.3 Inserting Text

Once the cursor is positioned, you can insert text in front of it simply by typing the text on the keyboard. No command is required and whatever you type becomes part of the file. Your insertion appears on the screen as you type it, and the surrounding text moves as necessary to accommodate it.

When you insert text at the beginning or in the middle of a line, the end of the line may disappear off the edge of the screen. The text is not lost, however: if you enter a carriage return in the text you are typing, the text appears on the next line. To avoid this problem, you can use the OPEN LINE function. When the cursor is at the beginning of a line, OPEN LINE provides a blank line above that line, and positions the cursor at the beginning of the blank line.

As you type new text, you may notice errors in surrounding text. You can move the cursor to these errors and correct them at any time, and then move the cursor back and continue to insert text.

2.5.4 Deleting and Undeleting Text

EDT character editing provides several methods of deleting text in units of varying sizes. EDT also maintains three buffers to contain text that has been deleted. The character buffer contains the last character deleted; the word buffer contains the last word deleted; and the line buffer contains the last line deleted. You can insert the contents of each of these three buffers at the cursor position by using the UND C, UND W, and UND L functions, respectively. There is no limit to the time or number of operations between a delete operation and the undelete operation that reinserts the deleted text. Furthermore, you can undelete one unit of text as many times as you wish, and at any locations you wish.

The DEL C (for character) function deletes the character at which the cursor is positioned, and moves the cursor to the next character. The

DELETE key on the keyboard deletes the character before the cursor position (the last character typed, if you are inserting text) and does not change the cursor position. Both of these functions move the deleted character into the character buffer, from which it can be retrieved by using the UND C function.

The DEL W (for word) function deletes from the current cursor position to (but not including) the first character of the next word. The LINE FEED key on the keyboard deletes from (but not including) the cursor position back to the first character of the current word. Both of these functions move the deleted text into the word buffer, from which it can be retrieved by using the UND W function.

The DEL L (for line) function deletes from the cursor position through the next end-of-line character. The DEL EOL (for end-of-line) function is similar, except that it does not delete the end-of-line character. Typing CTRL/U deletes from (but not including) the cursor position to the beginning of the current line. All of these functions move the deleted text into the line buffer, from which it can be retrieved by using the UND L function.

2.5.5 Moving Text

Character editing provides two basic methods of moving text. The first is available through the three undelete functions. You can delete a unit of text from one location, move the cursor to another location, and undelete the text there. However, this method is only effective for units that can be deleted by the various functions described in Section 2.5.4. To move larger or more precise blocks of text, use CUT and PASTE. These two functions allow you to "cut" any amount of contiguous text from one location and "paste" it elsewhere.

The first step is defining the text to be moved. To do this, move the cursor to either the beginning or end of the text and enter the SELECT function. Then move the cursor to the other extremity of the text. In so doing, you create a select range: that is, all the text between the cursor position and the position at which you entered the SELECT function. On VT100 terminals, EDT highlights the select range with reverse video. If you make a mistake while you are defining the select range, enter the RESET function to cancel the select range currently in effect.

Once you have defined the select range, enter the CUT function. The text within the select range disappears. (EDT moves it into a text buffer named PASTE.) Move the cursor to the position at which the text is desired, and enter the PASTE function. The text appears at the cursor position.

You can paste the cut text in as many locations as required. Specifically, you can paste the text as soon as you cut it, then move the cursor and paste the text again. This is in effect a copy operation.

Each CUT operation destroys the previous contents of the PASTE buffer and replaces them with the select range. To add the select range to the contents of the PASTE buffer, use the APPEND function.

The PASTE buffer is an ordinary EDT text buffer. You can edit within it, load it from a file with the INCLUDE command, and create a file from its contents with the WRITE command.

2.6 Protecting and Recovering Text

Three qualifiers to the EDIT/EDT command allow you to protect files against inadvertent modification and to recover editing operations that have been lost. This section discusses them.

The /READ_ONLY qualifier controls whether journaling and the creation of an output file are enabled. (Specifying /READ_ONLY is equivalent to specifying /NOOUTPUT and /NOJOURNAL.) /NOREAD_ONLY, the default, allows EDT to create an output file and a journal file. Use /READ_ONLY in situations where you want to be sure you do not create a modified file, or for reading a file in a directory where you do not have write privileges.

The /JOURNAL qualifier allows you to disable (using /NOJOURNAL) or to specify the name of the journal file that EDT creates to record your editing activity. By default, EDT creates a journal file with the file name of the input file and a file type of JOU. If the editing session ends abnormally, EDT can use the contents of the journal file to re-create the session. If the editing session ends normally (that is, as the result of an EXIT or QUIT command without a /SAVE qualifier), EDT deletes the journal file.

The /RECOVER qualifier causes EDT to use the contents of a journal file to re-create a previous editing session, perhaps one that was lost as the result of an accidental CTRL/Y or system problem. If you specify /RECOVER, EDT locates a file with the same name as the input file and a file type of JOU, then applies all the editing operations recorded in the journal file to the input file. These operations appear on your terminal as EDT performs them. When EDT has exhausted the contents of the journal file, the activity on the terminal ceases. You can now continue to edit.

Two notes of caution are necessary. First, it is important for the EDIT/EDT command that starts a recovery operation to match exactly the command that started the lost session, including any special startup command files. The only difference between the two commands should be the /RECOVER qualifier. In particular, the input file must be the same version that you started with at the beginning of the lost session. Second, note that EDT does not necessarily recover your session to the exact point where it was lost. A few keystrokes may be missing.

2.7 EDT Aids for the Programmer

In addition to the general-purpose editing operations discussed in Sections 2.1 through 2.6, EDT provides some advanced functions that are especially useful for programming. The following sections introduce some of these.

2.7.1 Structured Tabs

Although PL/I is a free-form language, in which excess spaces and tabs have no significance, it is common practice to indent lines to indicate the relationship of statements. It is laborious to enter repeatedly the correct combination of tabs and spaces to achieve the desired indention. EDT solves this problem by providing a system of structured tabs in character editing mode. While you are inserting text, a depression of the tab key inserts the correct combination of tabs and spaces to bring the cursor to the desired column. When you need to begin lines at a different column, you can increase or decrease the indention level to move the starting column to the left or right by a preset increment.

To use the structured tab feature, follow these steps:

1. While in line editing mode, set the increment between tabs by issuing the SET TAB command with a suitable value. For example:

   ```
   *SET TAB 4
   *
   ```

   At this point, the first TAB on a line (while in character editing mode) positions the cursor at column 5. Subsequent tab stops are at the normal locations.

2. When you want to change the indention level, use CTRL/E or CTRL/D. Each depression of CTRL/E increases the indention by one increment; the first tab stop is n spaces further to the right, where n is the number you gave with the SET TAB command. Pressing CTRL/D decreases the indention level.

3. If you want to set the indention level to correspond to a given column, position the cursor at that column and press CTRL/A. The column must be at an even multiple of n spaces from the left edge of the screen.

4. If you want to change the indention of a block of lines, first define a select range that includes the lines to be shifted. (To define a select range, position the cursor at one end of the block of lines, enter the SELECT function, and then position the cursor at the other end.) Then enter a repeat count (the GOLD key followed by a number typed on the keyboard) to indicate how many units of n spaces the

lines should be shifted. A positive repeat count shifts the lines to the right; a negative repeat count shifts the lines to the left. Finally, press CTRL/T.

2.7.2 Special-Purpose Key Definitions

EDT allows you to redefine the functions invoked by all the keys on the auxiliary keypad and many control characters as well. There are two ways to redefine a key's function:

- While in character editing mode, press CTRL/K. EDT prompts you to press the key you wish to define. Once you have pressed the key, EDT prompts you to enter the new function. You can do this either by typing the nokeypad commands that make up the function, or by pressing the keypad keys that correspond to the functions you require. You must follow the function specification with a period. The ENTER function terminates a definition of this type.
- While in line editing mode, issue the DEFINE KEY command. You define the new function to perform as a string of nokeypad character editing commands, followed by a period. The string and period must be enclosed in quotes.

Key redefinition requires a good grasp of nokeypad character editing syntax, as well as a good deal of practice. The EDT help facility (particularly HELP DEFINE KEY and HELP CHANGE SUBCOMMANDS) and the *VAX-11 EDT Editor Reference Manual* are good sources of information. However, this section describes one common application: the redefinition of a key to insert a string of text.

While writing a program, you may find that you are typing the same group of words over and over. For example, you might get tired of typing *PUT SKIP LIST*. In character editing mode, follow this procedure to define a key to insert the string *PUT SKIP LIST*:

1. Press CTRL/K. EDT prompts you with

   ```
   Press the key you wish to define
   ```

2. Select a function that you do not use often, for example, SPECINS. You might also select a control character. Enter the function or control character. EDT then prompts you with

   ```
   Now enter the definition terminated by ENTER
   ```

3. Type the following:

   ```
   iPUT SKIP LISTCTRL/Z.
   ```

   (The period is required syntax.)

4. Press ENTER to terminate the definition procedure.

For the remainder of the editing session, the key that used to invoke the SPECINS function will instead insert the string *PUT SKIP LIST* at the cursor position.

In line editing mode, you can redefine a key by using the DEFINE KEY command. To identify a keypad key in the command, you use a number. You can find out which numbers are assigned to which keys by issuing the command HELP DEFINE KEY VT52 or HELP DEFINE KEY VT100. These commands display the numbers assigned to keypad keys on the respective terminals.

Next, you issue a DEFINE KEY command, specifying the key and the function you wish the key to perform. The following example redefines the SPECINS function (GOLD/3 on a VT100) to insert the string *PUT SKIP LIST*:

```
*DEFINE KEY GOLD 3 AS "iPUT SKIP LIST^Z."
*
```

The quotes and period are required syntax. The ^Z is *not* a CTRL/Z, but a circumflex followed by a Z. For the remainder of the editing session, GOLD/3 will insert the string *PUT SKIP LIST* at the cursor position.

The preceding examples represent only a small fraction of the capabilities of key redefinition. With practice, you can create powerful custom functions that can save you a great deal of time. You may want to store these functions in a startup command file so that you will not have to define them each time you begin an editing session. The next section describes startup command files.

2.7.3 Startup Command Files

When you invoke EDT, it searches your current default directory for a file named EDTINI.EDT. If EDT finds such a file, it executes the line editing commands contained in the file before turning control over to you. This function allows you to customize EDT to suit your needs. Some of the commands that a startup command file might contain are

- DEFINE KEY. These commands redefine the function invoked by a keypad key or control character while in character editing mode. (See Section 2.7.2.)
- DEFINE MACRO. These commands associate a name with a sequence of line editing commands stored in a text buffer. You can then invoke the sequence by entering the macro name in response to the line editing asterisk prompt.

- INCLUDE. These commands bring text from a file into a text buffer. You might use them to load macros into a buffer, or to fill a buffer with text that you often use. (See Section 2.4.7.)
- SET. These commands establish EDT operating parameters. Particularly useful are SET TAB, which establishes the increment for structured tabs, and SET MODE CHANGE, which causes EDT to enter directly into character editing mode. (Section 2.7.1 describes the use of structured tabs.)

You can use the /COMMAND qualifier to the EDIT/EDT command to cause EDT to search for a file other than EDTINI.EDT. This means that you can have several startup command files, each designed for a particular application. You may want to include a command in your login command procedure file (see Section 1.4.2) to equate a short mnemonic to an EDIT/EDT command that invokes a special startup command file. For example, if you have the following line in your login command file:

```
$ EDP :== "EDIT/EDT/COMMAND=PLI.EDT"
```

then the command

```
$ EDP METRIC.PLI
```

invokes EDT with the startup command file PLI.EDT to edit the file METRIC.PLI.

Chapter 3

Compiling PL/I Programs

This chapter describes how to use the PLI command to compile your source programs into object modules. It discusses

- The functions of the compiler.
- PLI command syntax and qualifiers.
- The use of text libraries.
- Compiler diagnostic messages and error conditions.

3.1 Functions of the Compiler

The primary functions of the VAX-11 PL/I compiler are to verify the PL/I source statements and to issue messages if there are any errors; to generate machine language instructions from the source statements of the PL/I program; and to group these instructions into an object module for the linker.

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

- The module name. This is taken from the name of the main procedure in the source program, that is, the procedure that specifies OPTIONS (MAIN). If no procedure specifies OPTIONS (MAIN), the module name is the name on the first procedure statement in the source file.
- A list of all entry points and external 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. This 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, which aids program debugging.

- If specifically requested (with the /DEBUG qualifier), a symbol table. A symbol table lists the names of all external and internal variables within a module, with definitions of their locations. The table is of primary use in program debugging.

The linker is described in Chapter 4.

3.2 The PLI Command

The syntax of the PLI command and its qualifiers follows, as well as descriptions of the parameters and qualifiers. Subsequent sections give detailed examples and rules for specifying input and output files for the PLI command. The format is

```
PLI[/qualifier...] file-spec[/qualifier...],...
```

Command Qualifiers	**Defaults**
/[NO]CHECK	/NOCHECK
/CHECK[=option]	/CHECK=ALL
/[NO]CROSS_REFERENCE	/NOCROSS_REFERENCE
/[NO]DEBUG	/NODEBUG
/DEBUG[=option]	/DEBUG=ALL
/[NO]ERROR_LIMIT	/NOERROR_LIMIT
/ERROR_LIMIT[=n]	/ERROR_LIMIT=100
/[NO]G_FLOAT	/NOG_FLOAT
/[NO]LIST[=file-spec]	/NOLIST (interactive default) /LIST (batch default)
/[NO]MACHINE_CODE	/NOMACHINE_CODE
/MACHINE_CODE[=option]	/MACHINE_CODE=INTERSPERSED
/[NO]OBJECT[=file-spec]	/OBJECT
/[NO]OPTIMIZE[=(option,...)]	/OPTIMIZE=ALL
/SHOW[=(option,...)]	/SHOW=(NOINCLUDE, NODICTIONARY, NOMAP, SOURCE, NOTRACE, TERMINAL, NOEXPANSION, NOSTATISTICS)
/VARIANT[=["]alphanumeric_string["]	/VARIANT=""
/[NO]WARNINGS	/WARNINGS

File Qualifier

/LIBRARY

file-spec,...

Specifies one or more PL/I source files to be compiled and, optionally, libraries to be searched for INCLUDE files that are referenced in the source file(s).

You must separate multiple input file specifications with either commas (,) or plus signs (+). They have different meanings:

- Commas delimit PL/I source files to be compiled separately. PL/I compiles each file and creates an object module for each.
- Plus signs delimit files to be concatenated or libraries containing INCLUDE files. PL/I compiles the source files as a single file and creates one object module. Library file specifications must be qualified with the /LIBRARY qualifier.

If a file specification does not contain a file type, PL/I assumes a default file type of PLI for a source file. If a file specification is qualified with /LIBRARY, PL/I assumes a default file type of TLB. INCLUDE files and INCLUDE file libraries are described in Section 3.3 and Chapter 20.

A single file may contain multiple PL/I procedures; PL/I concatenates them into a single object module.

Command qualifiers request processing options of the compiler. You can specify qualifiers to the PLI command after the command name or an individual file specification. When a qualifier is specified after the PLI command name, its action applies to each file in the list, unless overridden by a qualifier specified for an individual file.

When a qualifier is specified after a file specification in a list of files separated by commas, its action is applied only to the compilation of that file.

/CHECK
/NOCHECK (default)

Controls the checking of array subscripts and of positional references in arguments to the SUBSTR built-in function. /CHECK is primarily of use during initial program debugging; it results in the generation of additional code and, consequently, a slower program.

Specifying /CHECK is equivalent to specifying /CHECK=ALL and /CHECK=BOUNDS. Likewise, /NOCHECK is the equivalent of specifying /CHECK=NONE and /CHECK=NOBOUNDS.

/CROSS_REFERENCE
/NOCROSS_REFERENCE (default)

Specifies whether the compiler is to generate, in the listing file, cross-references for variable names. If you specify /CROSS_REFERENCE, the compiler lists all variable names, including all members of structures as separate entities in an alphabetical cross-reference listing. The cross-reference entry for each structure member also lists the name of the structure that contains the member. The listing contains the line numbers of the lines on which all variables are referenced.

Note that /SHOW=MAP is required with /CROSS_REFERENCE.

By default, the compiler does not include cross-references in the listing.

/DEBUG[=option]
/NODEBUG

Requests that information be included in the object module for use with the VAX-11 Symbolic Debugger. You can select the following options:

ALL	Include symbol table records and traceback records. This is equivalent to /DEBUG.
SYMBOLS	Include symbol definitions for all identifiers. This is the default for symbols if the /DEBUG qualifier is used.
NOSYMBOLS	Do not include symbol definitions. Without symbol definitions, traceback is done according to virtual address.
TRACEBACK	Include only traceback records. This is the default if the /DEBUG qualifier is not present in the command.
NOTRACEBACK	Do not include traceback records.
NONE	Do not include any debugging information. This is equivalent to /NODEBUG. Use this option to exclude all debug information from thoroughly debugged program modules.

For an example of a traceback, see Section 5.1.2.

/ERROR_LIMIT[=n]
/NOERROR_LIMIT

VAX-11 PL/I permits you to specify the number of errors acceptable during program compilation. Normally, compilation terminates when the number of errors reaches 100, but /NOERROR_LIMIT raises this default number to 1000. However, you may specify a different error limit with the /ERROR_LIMIT=n qualifier. The maximum number of error messages permitted by the system is 32767.

All error and warning messages are counted toward the error limit. Fatal messages immediately terminate the compilation.

/G_FLOAT
/NOG_FLOAT (default)

For VAX-11 computers that are equipped with the appropriate hardware option, specifies the representation of floating-point variables with a binary precision in the range 25 through 53 and increases the

maximum precision available. By default, the compiler uses D (double-precision) floating point. Specify /G_FLOAT to override this default and to request the compiler to use the G floating-point type for these variables.

The default and maximum precisions for all floating-point formats are summarized in Section 8.2.3.

/LIST[=file-spec] (batch default)
/NOLIST (interactive default)

Controls whether a listing file is produced. When /LIST is in effect, the compiler gives a listing file the same file name as the source file and a file type of LIS. If you supply a file specification with /LIST, the compiler uses that file specification to override the default values applied.

You can control the contents of the listing file by specifying the /CROSS_REFERENCE and /MACHINE_CODE qualifiers, and by specifying options on the /SHOW qualifier.

/MACHINE_CODE[=option]
/NOMACHINE_CODE (default)

Controls whether the listing file produced by the compiler includes a listing of the machine code generated during the compilation.

You can select the following options:

AFTER	Put machine code after the source code.
BEFORE	Put machine code before the source code.
INTERSPERSED	Intersperse source and machine code.

/OBJECT[=file-spec] (default)
/NOOBJECT

Controls whether the compiler produces object modules. By default, the compiler produces an object module with the same file name as the source file and a file type of OBJ.

Specify /NOOBJECT when you want to compile a program to obtain only a listing or when you want the compiler to check the source program only for errors and display diagnostic messages. The compiler can execute more rapidly if it does not need to create an object module.

/OPTIMIZE[=(option,...)]
/NOOPTIMIZE

Controls the optimization performed by the compiler. By default, all possible optimizations are performed. The optimizations and the options that control them are described in the *VAX-11 PL/I User's Guide*.

If you specify /OPTIMIZE with any options, the settings of other options are not affected. For example, /OPTIMIZE=NOPEEPHOLE disables the PEEPHOLE option but leaves all other options enabled.

/SHOW[=(option,...)]

Sets or cancels specific compilation listing options. You can select or cancel any of the options listed in Table 3-1. The following options are enabled by default:

NOINCLUDE
NOMAP
NODICTIONARY
SOURCE
TERMINAL
NOSTATISTICS
NOTRACE
NOEXPANSION

The /SHOW qualifier must be used in combination with the /LIST qualifier before it can be effective. The /LIST qualifier specifies that a source listing is to be made, and the /SHOW qualifier gives you control over which portions of the source listing you want to see.

When you specify any option with the /SHOW qualifier, the settings for other options are not changed.

Table 3-1: PL/I Compiler Options

Option	Function
ALL	Include the contents of all files and modules in the program listing.
NONE	Do not include the contents of any of the files and modules in the program listing.
[NO]INCLUDE	Include/do not include the contents of INCLUDE files and modules in the program listing.
[NO]DICTIONARY	Include/do not include the contents of Common Data Dictionary record modules in the program listing.
[NO]MAP	Include/do not include the storage map of the compiled program in the program listing. The storage map includes a list of all external entry points, the size and attributes of all variables that are referenced in the program, and a program section summary and procedure definition map.
[NO]SOURCE	Include/do not include the source program statements in the program listing.
[NO]STATISTICS	Include/do not include performance statistics in the program listing.
[NO]TERMINAL	Display/do not display compilation messages to SYS$OUTPUT at compile time.

Table 3-1 (Cont.): PL/I Compiler Options

Option	Function
[NO]TRACE	Include/do not include each step of preprocessor replacement and rescanning.
[NO]EXPANSION	Include/do not include the final replacement values for preprocessor variables.

You can also control the content of the source listing by using preprocessor statements to suppress preprocessor portions in the program text. For example, if you previously specified /SHOW=INCLUDE, you may suppress included files from the listing with the %NOLIST_INCLUDE statement in your program.

By default, the /SHOW qualifier includes two listing notations specifically for preprocessor statements. An asterisk * in the column to the right of the line numbers indicates which portions of the program text were not used at compile time. A 'P' in the column to the right of the line numbers indicates that the preprocessor statement on that line is contained within a preprocessor procedure.

By default, the /SHOW qualifier yields a listing with two items (P and *) noted in the column to the right of the line numbers. However, additional items are noted depending on the value given to the qualifier. Table 3-2 summarizes the characters that can appear in the listing.

Table 3-2: Listing Notation Characters

Character	Qualifiers	Meaning
|	/LIST	Indicates a line that contains a comment only.
*	/LIST	Indicates program text that was not used at compile time.
D	/LIST/SHOW=DICTIONARY	Indicates CDD text included by a %DICTIONARY statement.
E	/LIST/SHOW=EXPANSION	Indicates the final replacement value of a preprocessor variable or procedure.
I	/LIST/SHOW=INCLUDE	Indicates text included by a %INCLUDE statement.
P	/LIST	Indicates lines contained within a preprocessor procedure.
T	/LIST/SHOW=TRACE	Indicates each step of preprocessor replacement and rescanning.

If you specify /LIST/SHOW=ALL, the compiler includes the full complement of character notations in the column to the right of the line numbers.

The effect of the /SHOW and /LIST qualifiers on the program listing is illustrated in the sample listings in Appendix A.

/VARIANT
/VARIANT""

Permits specification of compilation variants. The value specified for /VARIANT is available at compile time via the VARIANT() preprocessor built-in function.

If /VARIANT is not specified, or if /VARIANT is specified without a value, /VARIANT="" is assumed.

/WARNINGS (default)
/NOWARNINGS

Controls whether the compiler prints messages for diagnostic warnings. If you specify /NOWARNINGS, the compiler does not print warning messages. It does, however, continue to display messages for informational, error, and fatal diagnostics. (Section 3.4 contains more information about the significance of warning messages.)

File Qualifier

/LIBRARY

Indicates that the associated input file is a library containing text modules that may be included in the compilation of one or more of the specified input files. The specification of a library file must be preceded by a plus sign. If the file specification does not contain a file type, PL/I assumes the default file type of TLB.

For information on how the PL/I compiler locates text libraries, see Section 3.3. For information on creating INCLUDE file libraries, see Chapter 6.

3.2.1 PLI Command Examples

The following examples illustrate the use of the PLI command.

```
$ PLI METRIC
```

The PLI command compiles METRIC.PLI and creates the file METRIC.OBJ.

```
$ PLI/LIST/SHOW=INCLUDE/MACHINE_CODE APPLIC
$ PRINT APPLIC
```

The PLI command compiles the file APPLIC.PLI and creates the files APPLIC.OBJ and APPLIC.LIS. The listing shows the contents of all files

and text modules included in the compilation by %INCLUDE statements, as well as a machine code listing of the program. The /LIST qualifier is not necessary because /MACHINE_CODE implies /LIST. The PRINT command queues a copy of the listing file for printing. The default file type given to a listing file by the compiler is LIS; this is also the default file type assumed by the PRINT command.

```
$ PLI SWITCH.TXT/CHECK
```

The PLI command compiles the statements in the file SWITCH.TXT. The /CHECK qualifier causes the compiler to verify all array references and substring extents. The compiler produces the file SWITCH.OBJ.

The VAX-11 PL/I compiler lists the PLI command and its specified command qualifiers in the program listing.

3.2.2 Specifying Input and Output Files

To specify an alternative name for a listing or object file or an alternative target directory or device, you can include a file specification on the /LIST or /OBJECT qualifier. Some examples follow:

Command	Output File(s)
$ PLI METRIC/LIST=TEST	METRIC.OBJ (by default) TEST.LIS
$ PLI METRIC- $_/LIST=[PROJECT.LISTINGS]- $_/OBJECT=[PROJECT.OBJECT]]	 [PROJECT.LISTINGS] METRIC.LIS [PROJECT.OBJECT] METRIC.OBJ
$ PLI METRIC/LIST=LPA0:	METRIC.OBJ (by default) line printer listing
$ PLI/LIST=SYS$OUTPUT METRIC	METRIC.OBJ (by default) listing on the current output device

In the third and fourth examples, the listing files are not saved on disk; they are deleted after output.

3.3 Using Text Libraries

You can use text libraries to provide application-specific text modules within your particular environment. Chapter 6 contains information on creating text libraries. You gain access to modules in text libraries with the %INCLUDE statement.

The %INCLUDE statement (described in Section 7.4.3) provides a way for many separate programs to share common source text. For example, an application may consist of many separately compiled external procedures that share the same structure declaration or external variable declarations.

In such cases, it is convenient to maintain only one copy of the declaration of the variables and to include this declaration in each source program.

An %INCLUDE statement in a PL/I source file requests inclusion of an entire file, or of a module from a library of text files. When the compiler reads the %INCLUDE statement during compilation of a source program, it begins reading from the file or module specified by %INCLUDE. When it reaches the end of the included text, it resumes reading from the previous input file.

When an %INCLUDE statement in your program requests inclusion of a module from a library, you must be sure that the PL/I compiler can find the library. Either specify it explicitly in the PLI command, or request a module from one of the libraries that the compiler searches by default.

3.3.1 Specifying Text Libraries in the PLI Command

When you specify a library file in a PLI command, you must precede the specification with a plus sign and use the /LIBRARY qualifier. For example:

```
$ PLI APPLIC+DATAB/LIBRARY
```

This PLI command compiles the source program APPLIC.PLI and uses the library DATAB.TLB to locate any INCLUDE files that are referenced in the format

```
%INCLUDE text-module-name;
```

The module name must not be enclosed in apostrophes.

When you specify more than one library, PL/I searches the libraries in the order specified each time it processes an %INCLUDE statement that specifies a text module name. For example:

```
$ PLI APPLIC+DATAB/LIBRARY -
$_+NAMES/LIBRARY+GLOBALSYMS/LIBRARY
```

When PL/I processes an %INCLUDE statement in the source file APPLIC.PLI, it searches for modules referenced in the libraries DATAB.TLB, NAMES.TLB, and GLOBALSYMS.TLB, in that order.

On a command that requests multiple compilations, a library must be specified for each compilation in which it is needed. For example:

```
$ PLI METRIC+DATAB/LIBRARY,APPLIC+DATAB/LIBRARY
```

In this example, PL/I compiles METRIC.PLI and APPLIC.PLI separately and uses the library DATAB.TLB for each compilation.

The order of appearance of the library file specification within a concatenated list of files is irrelevant. For example, the following are equivalent:

```
$ PLI ALPHA+MYLIB/LIBRARY+BETA
$ PLI ALPHA+BETA+MYLIB/LIBRARY
```

3.3.2 Default PL/I Libraries

You can define one of your private INCLUDE file libraries as a default library for the PL/I compiler to search. The compiler searches the default library after it searches libraries specified on the PL/I command.

To define a default library, define an equivalence for the logical name PLI$LIBRARY, as in the following example:

```
$ DEFINE PLI$LIBRARY DATAB
```

While this assignment is in effect, the compiler automatically searches the library DATAB.TLB for any INCLUDE modules that it cannot locate in libraries explicitly specified on the PLI command.

You can define the logical name PLI$LIBRARY in the process, group, or system logical name table. If the name is defined in more than one table, the PL/I compiler uses the equivalence for the first match it finds in the normal order of search (that is, the process, then group, then system table). Thus, if PLI$LIBRARY is defined in both the process and group logical name tables, the process logical name table assignment overrides the group logical name table assignment.

When it cannot find INCLUDE modules in libraries specified on the PLI command or in the default library defined by PLI$LIBRARY, PL/I searches the library identified by the name

SYS$LIBRARY:PLISYSDEF.TLB

where SYS$LIBRARY is normally defined by the system manager to identify the device and directory containing system libraries. PLISYSDEF.TLB is a library of INCLUDE modules supplied by VAX-11 PL/I. It contains declarations for the entry points for VAX/VMS system services, local symbol definitions required for use with them, and variables to test their return status values.

3.4 Compiler Diagnostic Messages and Error Conditions

One of the functions of the PL/I compiler is to identify syntax errors and violations of language rules in the source program. If the compiler locates

any errors, it writes messages to your default output device; thus, if you enter the PLI command interactively, the messages are displayed on your terminal. If the PLI command is executed in a batch job, the messages appear in the batch job log file.

Each compilation with diagnostic messages terminates with a diagnostic summary that indicates the number of error, warning, and informational messages generated by the compiler. The diagnostic summary has the format

```
%PLIG-I-SUMMARY
     Completed with n error(s), n warning(s),
     n informational messages.
```

If the compiler creates a listing file, it also writes the messages to the listing. Messages typically follow the statement that caused the error.

When it appears on the terminal, a message from the compiler has the format

```
%PLIG-s-ident, message-text
          At line number n device:[directory]file.nme;x.
```

%PLIG

Is the facility, or program, name of the the VAX-11 PL/I Subset G compiler. This portion indicates that the message is being issued by PL/I.

s

Specifies the severity of the error. The letters that represent the possible severities are

F Fatal. The compiler stops executing, does not continue the compilation, and does not produce an object module. You must correct the error before you can compile the program.

E Error. The compiler continues, but does not produce an object module. You must correct the error before you can successfully compile the program.

W Warning. The compiler produces an object module. It attempts to correct the error in the statement, but you should verify that the compiler's action is acceptable. Otherwise, your program may produce unexpected results.

I Information. This message usually appears with other messages to inform you of specific actions taken by the compiler. Informational messages also indicate nonstandard constructs and items that are syntactically correct, but that may contain programming errors. No action is necessary on your part.

ident

Is the message identification. This gives a descriptive abbreviation of the message text.

message-text

Is the compiler's message. In many cases, the message text consists of more than one line of output. The messages generally provide enough information for you to determine the cause of an error and correct it.

At line number *n*

Specifies the source file line number of the statement that caused the error. This is the line number assigned to a statement by the compiler. It is not necessarily the same as the line number, if any, assigned by a text editing program.

device:[directory]file.nme;x.

Indicates its location, the filename, and version number.

The compiler produces messages with warning severity if it encounters

- Syntax errors (such as a missing END statement) that the compiler attempts to fix.
- Language elements (such as undeclared variables) that are not part of the PL/I G subset but do belong to full PL/I.
- Legal PL/I G subset usage (such as assignment of a bit-string value to a fixed-point binary variable) that nonetheless may represent a programming error or produce unexpected results.

3.5 User-Generated Diagnostic Messages

VAX-11 PL/I permits you to create and insert special-purpose preprocessor diagnostic messages to do the following:

- Write the message text.
- Specify the severity level.
- Define the condition which issues the message.

User-generated diagnostic messages are appropriate in the source program wherever a potential compile-time problem may develop or when specific compile-time information is required.

The following preprocessor statements, when used in conjuction with a %IF-group, specify the conditions that cause the diagnostic messages to be issued during program compilation:

```
%INFORM
%WARN
%ERROR
%FATAL
```

You determine the severity of the diagnostic message by your choice of statement. For example, if you wanted compilation information only, then you would use %INFORM. If you wanted to stop compilation where specific conditions developed, then you would use %FATAL.

The message text is specified by a preprocessor expression. The resulting message is returned in the same format as other compiler diagnostic messages.

When you determine the severity, you may also define the conditions which control the production of an object module. As with non-user-generated compiler messages, informational and warning messages do not inhibit the production of an object module. Error and fatal messages do. For example:

```
%IF SUBSTR(TIME(),1,2) > 7 & SUBSTR(TIME(),1,2) < 18
%THEN
    %FATAL 'Please compile this outside of prime time';
```

Here, the compiler aborts compilation if someone attempts to compile the program between the hours of 7 a.m. and 6 p.m., and it issues the following fatal diagnostic message:

```
%PLIG-F-USERDIAG, Please compile this outside of prime time
```

You can use preprocessor built-in functions to return—at specific points in the program—the number of diagnostic messages generated at compile time. For example, if you wanted to know how many warnings had been issued when compilation was half complete, you could insert the WARN preprocessor built-in function in the source program. Then, you could elect to terminate compilation if errors threaten successful compilation. For example:

```
%IF WARN() = 5
%THEN
    %FATAL 'Compilation aborted with 5 warnings';
%ELSE
    %;
```

See Chapter 20 for more information on the Embedded Preprocessor.

Chapter 4

Linking Programs

This chapter describes how to use the linker and object module libraries to combine object modules into executable programs. It discusses

- The functions performed by the linker.
- The LINK command and its input and output files.
- Object module libraries.

The topics in this chapter are confined to areas of particular interest to PL/I programmers. For additional information on linker capabilities and detailed descriptions of LINK command qualifiers and options, see the *VAX-11 Linker Reference Manual.*

4.1 Functions of the Linker

The primary functions of the linker are to allocate virtual memory within the executable image, to resolve symbolic references among modules being linked, to assign values to relocatable global symbols, and to perform relocation. The linker's end product is an executable image that you can run.

For any PL/I procedure, the object module generated by the compiler contains calls and references to VAX-11 PL/I run-time procedures, which the linker locates automatically in the default system object module libraries. The libraries are described in Section 4.3.

4.2 Using the LINK Command

The format of the LINK command is

```
LINK[/qualifier...] file-spec[/qualifier...],...
```

file-spec,...

Specifies one or more files containing object modules to be linked and, optionally, libraries containing modules that can be included. You can separate the file specifications with commas or plus signs. In either case, all files specified are used as linker input for the creation of a single executable image.

If the file specification does not contain a file type and is not qualified by /LIBRARY, /INCLUDE, or /OPTIONS, the linker assumes a default file type of OBJ.

/qualifier...

Specifies one or more LINK command qualifiers.

The /LIBRARY, /INCLUDE, and /OPTIONS qualifiers can be specified only after the specification of an input file. All other qualifiers can be specified either after the LINK command or after any input file specification. Table 4-1 summarizes the LINK command qualifiers in categories.

4.2.1 Linker Messages

If the linker detects any errors while linking object modules, it displays messages about their cause and severity. If any error or fatal conditions occur (severities E or F), the linker does not produce an image file.

Linker messages are descriptive, and you do not normally need additional information to determine the specific error. Some of the more common errors that occur during linking follow:

- An object module has compilation errors. This error occurs when you attempt to link a module that had warnings or errors during compilation. Although you can usually link compiled modules for which the compiler generated messages, you should verify that the modules will actually produce the output you expect.
- The modules that are being linked define more than one transfer address. The linker generates a warning if more than one module has an entry point designated with the OPTIONS (MAIN) keywords. The image file created by the linker in this case can be run; the entry point to which control is transferred is the first one that the linker found.
- A reference to a symbol name remains unresolved. This error occurs when you omit required module or library names from the LINK command and the linker cannot locate the definition for a specified global symbol reference.

If an error occurs when you link modules, you can often correct it simply by reentering the command string and specifying the correct modules or libraries.

Should an error indicate that a module with a name in the format PLI$_name cannot be located, you may not be linking the program with the correct PL/I run-time library. If you cannot locate or define the PL/I run-time library for any reason, check with your system manager or operator for information.

Table 4-1: LINK Command Qualifiers

Function	Qualifiers	Defaults
Request output files and define a file specification.	/EXECUTABLE[=*file-spec*]	/EXECUTABLE=*name*.EXE, where *name* is the name of the first input file.
	/HEADER	
	/PROTECT	
	/SHAREABLE[=*file-spec*]	/NOSHAREABLE
	/SYMBOL_TABLE[=*file-spec*]	/NOSYMBOL_TABLE
Request and specify the contents of a memory allocation listing.	/BRIEF	
	[NO]CONTIGUOUS	
	/[NO]CROSS_REFERENCE	/NOCROSS_REFERENCE
	/FULL	
	/P0IMAGE	
	/[NO]MAP	/NOMAP (interactive) /MAP=*name*.MAP (batch) where *name* is the name of the first input file.
	/[NO]SYSTEM[=*base address*]	
Specify the amount of debugging information.	/[NO]DEBUG	/NODEBUG
	/[NO]TRACEBACK	/TRACEBACK
Indicate that input files are libraries and to specifically include certain modules.	/INCLUDE=(*module-name*,...)	
	/LIBRARY	
	/SELECTIVE_SEARCH	
Request or disable the searching of default user libraries and system libraries.	/[NO]SYSLIB	/SYSLIB
	/[NO]SYSSHR	/SYSSHR
	/[NO]USERLIBRARY[=*table*]	/USERLIBRARY=ALL
Indicate that an input file is a linker options file.	/OPTIONS	

4.2.2 Linker Input Files

You can specify the object modules to be included in an executable image in any of the following ways:

- Specify files containing individual object modules created by a compiler. The linker assumes that any unqualified file specification is an object module.
- Specify one or more object module libraries to be searched to resolve references to external procedures and variables. These libraries are searched for all references that are not resolved among the modules specifically included in the compilation. You must qualify the file specification of the library with the /LIBRARY qualifier. Object module libraries are described in Section 4.3.
- Specify explicit modules in an object module library that are to be included in the image. You must qualify the file specification of the library with the /INCLUDE qualifier and specify the names of the desired object modules.
- Specify in a shareable image library explicit shareable images that are to be included in the image. You must qualify the file specification of the library with the /INCLUDE qualifier and specify the names of the desired shareable images.
- Specify an options file containing additional file specifications and special linker options. You must qualify the file specification of an options file with the /OPTIONS qualifier.

The linker uses the following default file types for input files:

File	File Type
Object module	OBJ
Library	OLB
Options file	OPT

The format and content of a linker options file are described in detail in the *VAX-11 Linker Reference Manual.* You may wish to use an options file if you have a very long list of input files to specify, if you want to link a module with a shareable image file, or if you want to request special linker options regularly.

4.2.3 Linker Output Files

When you enter the LINK command interactively and do not specify any qualifiers, the linker creates only an executable image file. By default, the image file has the same file name as the first or only object module specified and a file type of EXE. For example:

```
$ LINK A,B,C
```

This LINK command links the object modules in the files A.OBJ, B.OBJ, and C.OBJ and creates the image file A.EXE.

In a batch job, the linker creates both an executable image file and a storage map file by default. The default file type for map files is MAP.

To specify an alternative name for a map file or image file, or to specify an alternative output directory or device, you can include a file specification on the /MAP or /EXECUTABLE qualifier. Some examples follow.

Command	Output File(s)
$ LINK METRIC/MAP=TEST	METRIC.EXE (by default) TEST.MAP
$ LINK METRIC/EXE=[PROJECT.EXE]- $_/MAP=[PROJECT.MAP]	[PROJECT.EXE] METRIC.EXE [PROJECT.MAP] METRIC.MAP
$ LINK METRIC/MAP=LP:	METRIC.EXE (by default) line printer listing of the map file

In the third example, the map file is not saved on disk after it is printed.

4.3 Using Object Module Libraries

When they are linked, all PL/I programs use a system-supplied object module library containing routines that provide I/O and other system functions. However, you can use additional libraries to provide application-specific object modules within your particular environment. Chapter 6 contains information on creating such libraries.

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

1. Refer to the object module by name in your program in a CALL statement or function reference.
2. Make sure that the linker can locate the library containing the object module.

You specify that a linker input file is a library file by following it with 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 next sections describe the order in which the linker searches libraries that you specify explicitly, default user libraries, and system libraries.

4.3.1 Defining the Search Order for Libraries

You can specify as many libraries as you wish as input for the linker; 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 appear.

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.

4.3.2 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
```

LNK$LIBRARY is a logical name; 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 it. The logical names you must use are LNK$LIBRARY (as above), LNK$LIBRARY_1, LNK$LIBRARY_2, and so on, to LNK$LIBRARY_999. When more than one of these logical names exists during a link, the linker searches them in numeric order beginning with LNK$LIBRARY. The search order is as follows:

1. 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.

2. 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 is taken for each reference that remains unresolved.

4.3.3 Temporary Defaults for INCLUDE Files

The VAX-11 PL/I compiler uses the DCL rule for the application of temporary defaults in lists of input files, that is, for lists that are delimited with either commas or plus signs. For example:

```
$ PLI [INTRO.SRC]A,B
```

The directory specification above is applied to both B and A.

Temporary defaults also apply to file specifications in INCLUDE files; that is, if a file in an %INCLUDE statement does not specify a device and/or directory, the compiler uses the device and directory of the file in which the %INCLUDE statement was read. For example, if the file [INTRO.SRC]A contains the statement

```
%INCLUDE 'STATE';
```

the compiler attempts to locate [INTRO.SRC]STATE.PLI.

4.3.4 System Libraries

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

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

The library IMAGELIB.OLB contains references to entry points in VMSRTL.EXE, which contains the VAX-11 Run-Time Library, and to entry points in PLIRTL.EXE, which contains the PL/I run-time library. The procedures in these libraries provide

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

IMAGELIB.OLB is a shareable image library; that is, it is prelinked and can be accessed by many images concurrently. The procedures in a shareable image library can be used by a program even though the procedures

are not physically included in the program image; the references to the procedure in the shareable image library are not resolved until the program is run.

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

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

4.3.5 Creating Shareable Images

You can create a shareable image that resides in your directory and that you can include in your applications by using the /SHAREABLE qualifier on the LINK command. The resulting image is stored as an executable image, but must be included in another program before it can be executed. The format is

```
$LINK/SHAREABLE file-spec
```

This command creates an individual shareable image that can be copied from your directory to that of another user.

Note that the concept of a shareable image includes the idea that a single copy of an image is created and stored in system memory so that many applications may use the same shareable image and save space by doing so. If you create a shareable image that resides in your directory and copy it to the directory of another user, you have in fact created two copies of the image and have defeated the purpose of shareable images.

The INSTALL utility allows you to create a single shareable image that can be shared across the system. Use of this utility conserves disk storage space and main physical memory, reduces paging I/O, and preserves the integrity of memory-resident data bases. For details on the use of the INSTALL utility, see the *VAX-11 Guide to Creating Modular Library Procedures*, Chapter 7, Building Modular Procedure Libraries.

Chapter 5

Running PL/I Programs on VAX/VMS

This chapter describes the following considerations for executing your PL/I programs on the VAX/VMS operating system:

- Using the RUN command to execute programs interactively
- Passing status values to the command interpreter

5.1 The RUN Command

You execute a PL/I program with the RUN command. The default file type for RUN is EXE, so you need not specify it. For example:

```
$ RUN METRIC
```

This RUN command locates the file METRIC.EXE in the current default directory. It then gives control to the main entry point, that is, the entry point designated with the OPTIONS (MAIN) keywords on its PROCEDURE or ENTRY statement. If no procedure specifies OPTIONS (MAIN), then control is given to the first or only module in the image.

5.1.1 Image Exit

When the main procedure executes a RETURN or END statement, or when any procedure in the program executes a STOP statement, the image is terminated. In the VAX/VMS operating system, the termination of an image, or image exit, causes the system to perform a variety of cleanup operations during which open files are closed, system resources are freed, and so on.

In a PL/I program, you can define an ON-unit to receive control when image exit occurs by executing an ON statement for the FINISH condition. For an example, see Section 15.1.5.

5.1.2 Run-Time Errors

When an error occurs during the execution of a program, and no ON-unit exists to handle the error, the program is terminated and one or more messages are displayed on the current SYS$ERROR device. The message(s) may actually be generated by one of the following:

- The default PL/I ON-unit, or, in VAX/VMS terms, condition handler. This condition handler exists if one of the procedures in the program was compiled with OPTIONS (MAIN).
- A default error condition handler established by the command interpreter.

In either case, the message is followed by a traceback. For each module having traceback information, the default handler lists the procedures that were active when the error occurred and the sequence in which the procedures were called, that is, the order of block activation.

For example, if an integer divide-by-zero condition occurs, and no ZERODIVIDE ON-unit exists in any active procedure block, the following run-time messages appear:

```
%PLI-F-ERROR, PL/I ERROR condition signaled
%SYSTEM-F-FLTDIV-F, arithmetic fault, floating divide
by zero at PC=000007C4, PSL=03C000A5
```

These messages are followed by a traceback message like the following:

```
%TRACE-F-TRACEBACK, symbolic stack dump follows

module      routine
name        name         line    relative PC    absolute PC

SETUP       DIVIDE         9     00000074       000007C4
SETUP       BEGIN%4        4     00000035       00000707
SETUP       SETUP          4     0000000C       000006D0
LIBS        NEXT          14     00000044       000006A3
LIBS        LIBS          15     0000004C       0000065E
```

These columns provide information as described below.

module name

Indicates the name of a level-1 procedure that was active when the error occurred. The first module is that in which the error occurred. Each subsequent line names the caller of the procedure on the previous line. In this example, the level-1 procedures are LIBS and SETUP; a call to SETUP occurred during the execution of LIBS.

routine name

Indicates the entry name of the internal procedure or block in the calling sequence. When BEGIN %*n* appears in this column, it indicates that an unlabeled begin block, a PUT statement, or a GET statement was active when the error occurred.

PL/I assigns labels to these blocks, giving them names in this form, where *n* is the source program line number on which the block is entered.

line

Indicates the compiler-generated source program line number of the statement at which the error occurred, or at which the call or reference to the next procedure was made. This line number matches those on the listing file created if /LIST was specified to the compiler.

relative PC

Gives the value of the PC (program counter).

absolute PC

Gives the value of the PC in absolute terms.

5.1.3 Interrupting a Program

When you execute the RUN command interactively, you cannot execute any other program images or DCL commands until the current image completes. However, if your PL/I program is not performing as expected—if, for instance, you have reason to believe it is in an endless loop—you can interrupt it. To do so, use CTRL/Y. (You may also use CTRL/C, unless your program takes specific action in response to CTRL/C.) For example:

```
$ RUN APPLIC
^Y
$
```

This command interrupts the program APPLIC. After you have interrupted a program, you can terminate it by entering a DCL command that causes another image to be executed or by entering the DCL command EXIT. PL/I signals the FINISH condition to allow a FINISH ON-unit to execute before the given DCL command is executed. You can also issue the DCL STOP command, which terminates the program and does not give control to the FINISH ON-unit.

Following a CTRL/Y interruption, you can also force an entry to the debugger by entering the DEBUG command.

There are some other DCL commands you can enter that have no direct effect on the image. After using them, you can resume the execution of the image with the DCL command CONTINUE. For example:

```
$ RUN APPLIC
^Y
$ SHOW TRANSLATION INFILE
  INFILE =     (undefined)
$ DEFINE INFILE DBA1:[TESTFILES]JANUARY.DAT
$ CONTINUE
```

For a complete list of the commands you can enter following a CTRL/Y interruption without affecting the current image, see the *VAX/VMS Command Language User's Guide.*

As noted above, you may use CTRL/C to interrupt your program; in most cases, the effect of CTRL/C and CTRL/Y is the same. However, some programs (including programs you may write) establish particular actions to take to respond to CTRL/C . If a program has no CTRL/C handling routine, then CTRL/C is the same as CTRL/Y and in fact is echoed as ^Y on the terminal.

5.2 Returning Status Values to the Command Interpreter

You can define a main procedure to be executed under the control of the DCL command interpreter as a PL/I function. Then the RETURN statement that terminates the main procedure can specify a status value to be used as a success, failure, or informational indicator to the command interpreter. For example:

```
TESTP: PROCEDURE OPTIONS (MAIN)
       RETURNS (FIXED BINARY(31));
           .
           .
           .
   RETURN (value);
```

where the value specified on the RETURN statement can be any constant, variable, or expression convertible to a fixed-point binary value. For meaningful results, you must specify the returns descriptor on the RETURNS option for the PROCEDURE statement as FIXED BINARY (31).

When the command interpreter receives a status value from a terminating program, it attempts to locate a corresponding message in a central system message file or a user-defined message file. Every possible message that can be issued by a system program, command, or component, has a unique 32-bit numeric value associated with it.

If you write a main procedure that returns arbitrary values, the command interpreter may use them to display messages that you would not expect. On the other hand, you may take advantage of this convention and use the RETURN statement to exit from a program's error-handling routine by specifying the status value associated with the error. For an example of this technique, see Section 15.1.5.

The command interpreter does not display messages on completion of a program under the following circumstances:

- A RETURN statement specifies the value 1, corresponding to SUCCESS.
- The procedure does not return a value. If the main procedure is not declared with the RETURNS option, a value of 1 is always returned and no message is displayed.
- The procedure executes a STOP statement.

Chapter 6

Creating Libraries

VAX/VMS and VAX-11 PL/I allow you to build, maintain, and use the contents of two kinds of library: text and object module. Text libraries contain modules of source text that you can include in a program by using an %INCLUDE statement. Object module libraries—both user-written and system-supplied—contain compiled code that the linker incorporates into an image to satisfy unresolved references.

This chapter covers the use of the LIBRARY command to create and maintain text and object module libraries.

6.1 The LIBRARY Command

The following description of the LIBRARY command does not include all the available qualifiers, only those that are useful to the PL/I programmer. For a complete description of the command, see the *VAX/VMS Command Language User's Guide*. The DCL HELP command also provides information about LIBRARY command qualifiers and functions not covered here.

The format of the LIBRARY command is

```
LIBRARY library-file-spec [input-file-spec[,...]]
```

Qualifiers with No Default

```
/COMPRESS [=option[,...]
/CREATE[=(option[,...])
/CROSS_REFERENCE[=(option[,...])
/DELETE=(module[,...])
/EXTRACT=(module[,...])
/FULL
/HELP
/INSERT
/MACRO
/OBJECT
/ONLY=(module[,...])
/OUTPUT=file-spec
```

Qualifiers with No Default

/REMOVE=(symbol[,...])
/REPLACE
/SELECTIVE_SEARCH
/TEXT
/WIDTH=n

Qualifiers	Default
/[NO]GLOBALS	/GLOBALS
/[NO]LIST[=file-spec]	/NOLIST
/[NO]LOG	/NOLOG
/[NO]NAMES	/NONAMES
/[NO]SQUEEZE	/SQUEEZE

library-file-spec

Gives the name of the library you want to create or modify. If the file specification does not include a file type, the LIBRARY command assumes a default type of OLB, indicating an object library.

input-file-spec[,...]

Gives the names of one or more files that contain modules you want to insert into the specified library.

Whenever you include an input file specification, the LIBRARY command either replaces or inserts the modules contained in the input file(s) in the specified library. The input-file-spec parameter is required when you specify either /REPLACE (the LIBRARY command's default operation) or /INSERT (an optional qualifier).

When you use the /CREATE qualifier to create a new library, the input-file-spec parameter is optional. If you include it with /CREATE, the library command first creates a new library and then inserts the contents of the input file(s).

Note that the /EXTRACT qualifier does not accept an input file specification.

If any file specification does not include a file type, the LIBRARY command assumes a default file type of OBJ, designating an object library. You can control the default file type by specifying the appropriate qualifier as indicated below:

Qualifier	Default File Type
/HELP	HLP
/MACRO	MAR
/OBJECT	OBJ
/TEXT	TXT
/SHARE	EXE

/COMPRESS[=(option[,...])]

Requests the LIBRARY command to recover unused space in the library resulting from module deletion or to reformat a library.

Options override values specified on creation:

BLOCKS:n
GLOBALS:n
HISTORY:n
KEEP
KEYSIZE:n
MODULES:n
VERSION:n

/CREATE

Requests the LIBRARY command to create a new library. You may optionally specify a file or a list of files that contains modules to be placed in the library. By default, the LIBRARY command creates an object module library; specify /TEXT to change the library type to a text library.

/CROSS_REFERENCE[=(option[,...])]

Requests a cross reference listing of an object library.

Options:

ALL	
MODULE	Global symbol definitions and references
NONE	
SYMBOL	Symbols by name
VALUE	Symbols by value

/DELETE=(module[,...])

Requests the LIBRARY command to remove the specified module(s) from the library.

/EXTRACT=(module[,...])

Copies one or more modules from an existing library into a new file. By default, the /EXTRACT qualifier copies the modules into a file that has the same file name as the library and a type of OBJ, MAR, or HLP. TXT. Use the /OUTPUT qualifier to override this default.

/FULL

Requests a full description of each module in the module name table. Use this qualifier in conjunction with /LIST.

/GLOBALS
/NOGLOBALS

Controls for object module libraries, whether the names of global symbols in modules being inserted in the library are included in the global symbol table.

/HELP

Indicates that the library is a help library. When you specify the /HELP qualifier, the library file type defaults to HLB and the input file type defaults to HLP.

/INSERT

Requests the LIBRARY command to add the contents of one or more files to an existing library. If a module name or global symbol name is already in the library, the command issues an error message and does not add the module.

/LIST[=file-spec]
/NOLIST (default)

Controls whether or not the LIBRARY command creates a listing of the contents of the library. If you specify /LIST without a file specification, the listing appears on the current SYS$OUTPUT device. If you include a file specification that has no file type, the LIBRARY command uses the default file type of LIS.

/LOG
/NOLOG

Controls whether the LIBRARY command verifies each library operation. If you specify /LOG, the LIBRARY command displays the module name, followed by the library operation performed, followed by the library file specification.

/MACRO

Indicates that the library is a macro library. When you specify /MACRO, the library file type defaults to MLB and the input file type defaults to MAR.

/NAMES
/NONAMES

Controls when /LIST is specified for an object module library, whether the LIBRARY command lists the names of all global symbols in the global symbol table as well as the module names in the module name table.

The default is /NONAMES, which does not list the global symbols names.

/OBJECT

Indicates that the library is an object module library. This is the default condition. The LIBRARY command assumes a library file type of OLB and an input file type of OBJ.

/ONLY=(module[,...])

Specifies the individual modules on which the LIBRARY command may operate. When you use the /ONLY qualifier, the LIBRARY command lists or cross references only those modules specified.

/OUTPUT=file-spec

With the /EXTRACT qualifier, specifies an output file to contain the modules extracted from a library. If you do not include a file type, the default is OBJ for modules extracted from object libraries and TXT for modules extracted from text libraries.

/REMOVE=(symbol[,...])

Requests the LIBRARY command to delete global symbol entries from the global symbol table in an object library.

/REPLACE

Requests the LIBRARY command to replace one or more existing library modules with those specified in the input file. If any module contained in the input file does not have a corresponding module in the library, the LIBRARY command inserts it. /REPLACE is the LIBRARY command's default operation.

/SELECTIVE_SEARCH

Defines the input files being inserted into a library as candidates for selective searches by the linker. If you specify /SELECTIVE_SEARCH, the linker selectively searches the modules when the library is specified as a linker input file; the linker only indicates the global symbol(s) in the module(s) referenced by other modules in the symbol table of the output image file.

/SQUEEZE
/NOSQUEEZE

Controls whether the LIBRARY command compresses individual macros before adding them to a macro library.

/TEXT

Indicates a text library. When you use the /TEXT qualifier, the library file type defaults to TLB and the input file type to TXT.

/WIDTH=n

Controls the screen width (in characters) when /NAMES is specified.

File Qualifier

/MODULE=module-name

Specifies the module name of a text module. By default, text libraries use the file name from the input-file-spec parameter as the module name. Use the /MODULE qualifier if you want to override this default.

6.2 Creating and Correcting Text Libraries

A text library is a file that contains individual files and a table indexing them. The LIBRARY command creates and modifies text libraries, which have a default file type of TLB. To use libraries for PL/I INCLUDE files, you must

1. Create one or more libraries consisting of INCLUDE files.
2. Specify the name of the INCLUDE module in an %INCLUDE statement in the PL/I source program.
3. Specify the name of the library on the PLI command to compile the source program or define a default user library.

Figure 6–1 illustrates the creation of an INCLUDE file library and its use in compiling PL/I programs. When the LIBRARY command adds a module to a library, it uses by default the file name of the input file as the name of the module. In the example in Figure 6–1, the LIBRARY command adds the contents of the files APPLIC.SYM and DECLARE.PLI to the library and names the modules APPLIC and DECLARE.

Alternatively, you can specify a name to be given a module in a library with the /MODULE qualifier. For example:

```
$ LIBRARY/TEXT/INSERT PLIFILES -
$_DECLARE.PLI/MODULE=EXTERNAL_DECLARATIONS
```

This command inserts the contents of the file DECLARE.PLI in the library PLIFILES under the name EXTERNAL_DECLARATIONS. This module can be included in a PL/I source file during compilation with the statement

```
%INCLUDE EXTERNAL_DECLARATIONS;
```

You can correct a module in a text library by following these steps:

1. Extract the module from the library by using the /EXTRACT qualifier to the LIBRARY command. Use the /OUTPUT qualifier to place the module in a file.
2. Make the necessary corrections by editing the file.
3. Replace the module in the library.

For example:

```
$ LIBRARY/TEXT/EXTRACT=(EXTERNAL_DECLARATIONS) -
$_PLIFILES /OUTPUT=TEMP.PLI
$ EDIT TEMP.PLI
      .
      .
      .
$ LIBRARY/TEXT PLIFILES TEMP.PLI-
$_/MODULE=EXTERNAL_DECLARATIONS
```

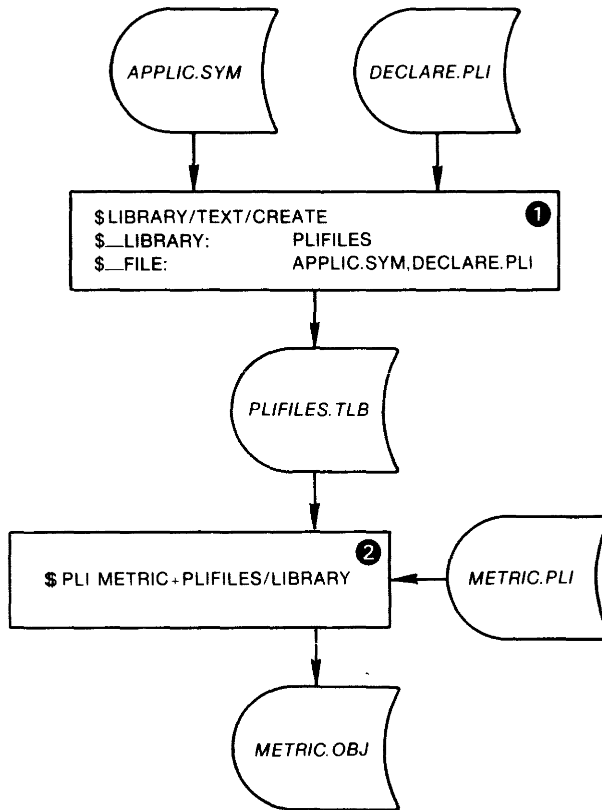


Figure 6-1: Creating and Using an INCLUDE File Library

6.3 Creating and Correcting Object Module Libraries

An object module library is a single file containing individual object modules and two tables that index the modules:

1. A module name table lists the names of the modules in the library. The names are those given at compilation.
2. A global symbol table lists all global symbols defined in each module.

These are the tables that are searched by the linker.

You can use object module libraries to

- Catalog and group together commonly used subroutines and functions.
- Provide a default set of modules for the linker to use in resolving global references in object modules it is linking.
- Enhance the performance of linking operations by putting all needed modules in a single library, thus reducing the number of files that need to be opened during the linking.

Figure 6–2 illustrates the sequence of creating object modules, creating a library, and using the library in linking programs.

The LIBRARY command uses the following default file types:

- OLB for an object module library file
- OBJ for an object module file

When the LIBRARY command inserts an object module in a library, it

- Enters the name of the module in the library's module name table.
- Enters all global symbols from the object module, including the names of all entry points and all variables designated as global symbols, in the library's global symbol table.

For example, a PL/I program named QUEUES.PLI might contain the following designations:

```
READY:  PROCEDURE;
    .
    .
    .
ADDEL:  ENTRY (QUEUE,POINTER);
    .
    .
    .
REMVEL: ENTRY (QUEUE,POINTER);
```

This module can be compiled and placed in a library as follows:

```
$ PLI QUEUES
$ LIBRARY/INSERT DEFLIB QUEUES
```

After this LIBRARY command, the module name table for the library DEFLIB.OLB contains an entry for the module named READY, and the library's global symbol table contains entries for the names ADDEL, REMVEL, and READY. Object modules that refer to any of those names can be linked with this library. When the library is specified as input to the linker, it searches both tables for unresolved references.

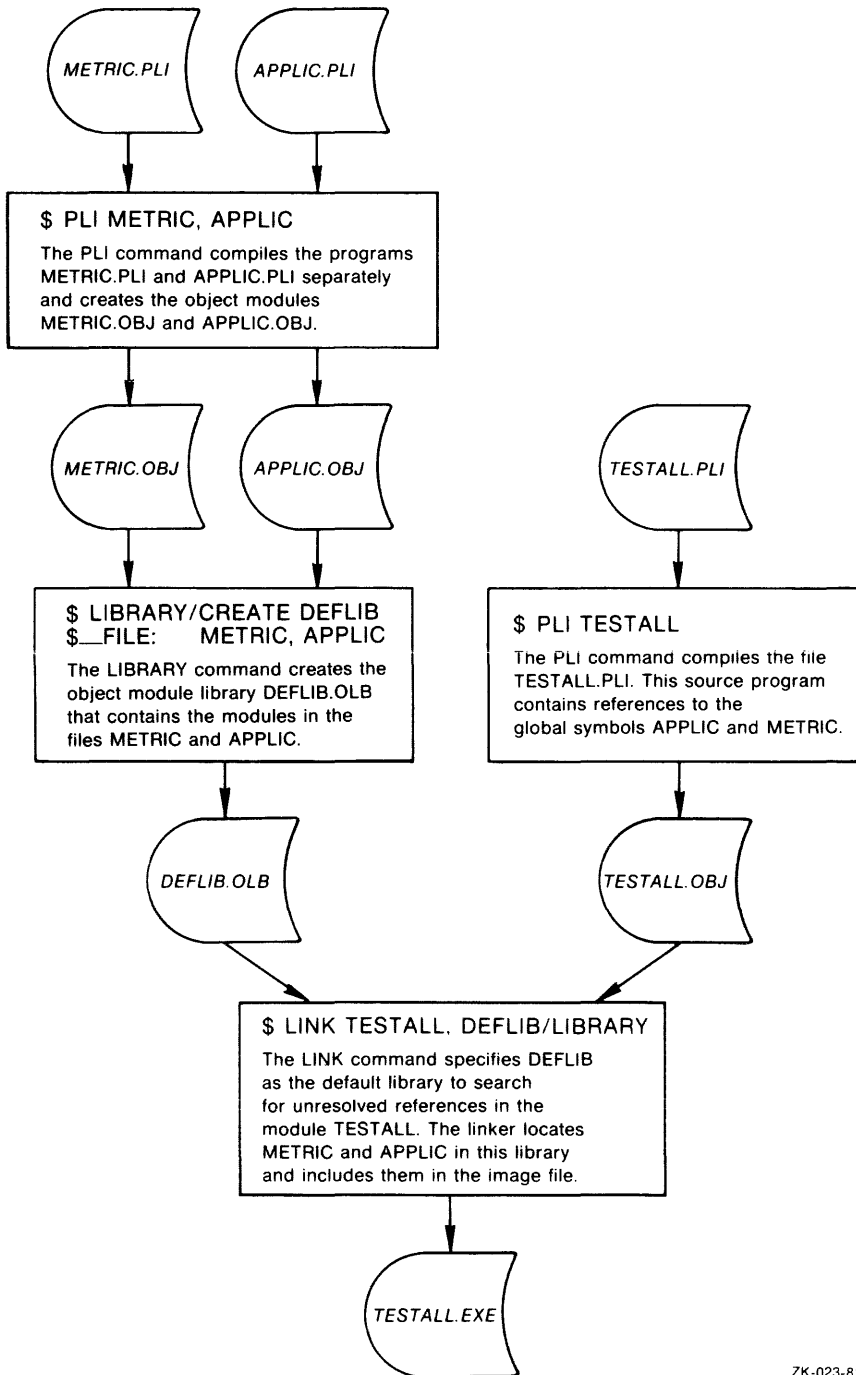


Figure 6-2: Creating and Using an Object Module Library

You can correct a module in an object module library by correcting the source file, compiling it, and then using the LIBRARY command to replace the module in the library. The following example shows commands you could use to correct a module in DEFLIB.OLB:

```
$ EDIT QUEUES.PLI
      .
      .
      .
$ PLI QUEUES
$ LIBRARY DEFLIB QUEUES
```

Chapter 7

Program Structure and Content

This chapter introduces and summarizes the elements of a PL/I program:

- Section 7.1 describes the blocks that make up a program and their effect during program execution.
- Section 7.2 describes the elements that make up a PL/I statement and lists the statements available in VAX-11 PL/I.
- Section 7.3 describes PL/I data types and lists the VAX-11 PL/I data type attributes.
- Section 7.4 discusses the text of a PL/I program.

Subsequent chapters treat these subjects in more detail.

7.1 Blocks

PL/I is a block-structured language: statements are grouped into blocks. There are two types of blocks: procedure blocks and begin blocks. A procedure executes only as the result of a specific request from another procedure or, in the case of the main procedure, as the result of a RUN command. A begin block is always contained within a procedure, and executes when control flows into it.

When control passes to either type of block, a block activation is created for it. A block activation consists of the allocation of storage for some of the variables declared within the current block and information that connects the current block to the previous one.

7.1.1 Begin Blocks

A begin block is a sequence of statements headed by a BEGIN statement and terminated by an END statement. Generally speaking, you can use a begin block wherever a single PL/I statement would be valid. In some

contexts, such as an ON-unit, a begin block is the only way to perform several statements instead of one. Another primary use of begin blocks is to localize variables. Since execution of a begin block causes a block activation, automatic variables declared within the begin block are local to it, and their storage disappears when the block completes execution.

Another way to allow your program to perform several statements in place of one is to use a DO-group (see Section 14.1.1). You should choose it when possible because it does not incur the overhead associated with block activation. Use a a begin block when there are declarations present or you require multiple statements in an ON-unit.

Section 14.2 contains the syntax for the BEGIN statement and examples of begin blocks.

7.1.2 Procedures

A procedure is a sequence of statements (perhaps including begin blocks and other procedures) headed by a PROCEDURE statement and terminated by an END statement. Unlike a begin block, which executes when control reaches it, a procedure executes only when it is specifically invoked. Invocation occurs in two ways:

- The DCL RUN command invokes the main procedure of a PL/I program. This is either the procedure that has OPTIONS (MAIN) on its PROCEDURE statement or the first procedure encountered by the linker.
- Statements within a procedure can invoke other procedures. The CALL statement invokes a procedure as a subroutine. A function reference invokes a procedure to return a value for use in the evaluation of an expression.

A PL/I program must have at least one procedure, the main procedure. Any procedure, including the main procedure, can contain others; these are called internal procedures. A procedure that is not contained within any other is called an external procedure. Note that the main procedure is therefore an external procedure.

Except for the main procedure, no procedure executes unless it is invoked by a CALL statement or function reference.

See Chapter 13 for more detailed information.

7.2 Statements

A statement is the basic element of a PL/I procedure. Statements are used to

- Define and identify the structure of the program and the data that it acts upon.
- Request specific actions to be performed on data.
- Control the flow of execution in a program.

In this manual, each PL/I statement is described in the chapter that covers the function associated with it. The description of each statement gives its syntax, abbreviation, if any, and options.

The general format of a PL/I statement consists of an optional statement label, the body of the statement, and a required terminator, the semicolon (;).

7.2.1 Statement Labels

A label identifies a statement so that the statement can be referred to elsewhere in the program, for example, as the target of a GOTO statement. A label precedes its statement; it consists of any valid identifier (see Section 7.2.3) terminated by a colon. For example:

```
TARGET: A=A+B;
READ_LOOP: READ FILE (TEXT) INTO (TEMP);
```

No statement can have more than one label.

7.2.2 Keywords and Punctuation

A keyword is a name that has a special meaning to PL/I when used in a specific context. In context, keywords identify statements, attributes, and options.

You can abbreviate some PL/I keywords. The valid abbreviations for PL/I keywords are given with the keyword description in this manual.

PL/I also recognizes punctuation marks in statements. The punctuation marks serve to

- Specify arithmetic or relational operations to be performed on expressions (see Section 12.2 for details).
- Delimit and separate identifiers, keywords, and constants.

For example, in the statement

```
A = B + C;
```

the equal sign (=), representing the assignment statement, the addition operator (+), and the semicolon (;) delimit the identifiers A, B, and C, as well as defining the operation to be performed. (Section 12.2 describes the effect of the various operators in expressions.)

Whenever you use a punctuation mark in a PL/I statement, you can precede or follow the character with any number of spaces. For example, the following two statements are equivalent:

```
DECLARE (A,B) FIXED DECIMAL (7,0);
DECLARE(A,B)FIXED DECIMAL(7,0);
```

In the second statement, the spaces preceding and following parenthetical expressions are omitted; the parentheses themselves are sufficient to distinguish elements in the statement. The only space required in this statement is the space that separates the two keywords FIXED and DECIMAL.

Table 7-1 summarizes the punctuation marks that PL/I recognizes. Note that operators consisting of two characters (for example, ** and >=) must be entered without intervening spaces in a PL/I program.

Table 7-1: Punctuation Marks Recognized by VAX-11 PL/I

Category	Symbol	Meaning to PL/I
Arithmetic operators	+	Addition or prefix plus
	-	Subtraction or prefix minus
	/	Division
	*	Multiplication
	**	Exponentiation
Relational (or comparison) operators	>	Greater than
	<	Less than
	=	Equal to
	^>	Not greater than
	^<	Not less than
	^=	Not equal to
	≥	Greater than or equal to
	≤	Less than or equal to
Logical operators	^	Logical NOT
	&	Logical AND
	¦ or !	Logical OR
Concatenation operator	¦¦ or !!	String concatenation

Table 7-1 (Cont.): **Punctuation Marks Recognized by VAX-11 PL/I**

Category	Symbol	Meaning to PL/I
Separators	,	Delimits elements in a list
	;	Terminates a PL/I statement
	.	Separates identifiers in a structure name; specifies a decimal point
	:	Terminates a procedure name or a statement label; separates elements of a bound pair in an array declaration
	( )	Enclose lists and extents; define the order of evaluation of expressions; separate statement and option names from specific keywords
	'	Delimit character strings and bit strings
Locator qualifier	->	Pointer resolution

In addition to punctuation marks, PL/I accepts spaces, tabs, and line-end characters between identifiers, constants, and keywords.

The line-end character is a valid punctuation mark between items in a PL/I statement except when it is embedded in a string constant, where it is ignored. For example:

```
A = 'THIS IS A VERY LONG STRING THAT MUST BE CONTI
NUED ON MORE THAN ONE LINE IN THE SOURCE FILE' ;
```

This assignment statement gives the variable A the value of the specified character-string constant, ignoring the line-end character. Note, however, that any tabs or spaces preceding *NUED* in the example above will be included in the string.

7.2.3 Identifiers

An identifier is a user-supplied name for a procedure, a statement label, or a variable that represents a data item.

7.2.3.1 Rules for Identifiers

The rules for forming identifiers are

- An identifier can have from 1 to 31 characters.
- An identifier can consist of any of the following characters:
  - The alphabetic letters A through Z and a through z. PL/I converts all lowercase letters to uppercase when it compiles a source program. Thus, the identifiers abc, ABC, Abc, and so on all refer to the same object.

  - The numeric digits 0 through 9.
  - The underscore character (_).
  - A dollar sign character ($).
- An identifier cannot contain any blanks.
- An identifier must begin with an alphabetic letter, a dollar sign ($), or an underscore (_).

Some examples of valid identifiers are

```
STATE
total
FICA_PAID_YEAR_TO_DATE
ROUND1
SS$_UNWIND
```

7.2.4 Alphabetic Summary of Statements

Table 7-2 provides an alphabetic summary of PL/I statements, and identifies the section that contains their descriptions.

Table 7-2: Summary of VAX-11 PL/I Statements

Statement	Use	Section
assignment	Evaluates an expression and gives its value to an identifier	12.1
null	Specifies no operation	14.9
ALLOCATE	Allocates storage for a based or controlled variable	9.5.3
BEGIN	Denotes the beginning of a block of statements to be executed as a unit	14.2
CALL	Transfers control to a subroutine or external procedure	13.1.4
CLOSE	Terminates association of a file control block with an input or output file	16.1.3
DECLARE	Defines the variable names and identifiers to be used in a PL/I program and specifies the data attributes associated with them	11.1
DELETE	Removes an existing record from a file	18.1.4
DO	Denotes the beginning of a group of statements to be executed as a unit	14.1
END	Denotes the end of a block or group of statements begun with a BEGIN, DO, SELECT, or PROCEDURE statement	14.3
ENTRY	Specifies an alternate point at which a procedure can be invoked	13.1.3

Table 7-2 (Cont.): Summary of VAX-11 PL/I Statements

Statement	Use	Section
FORMAT	Specifies the format of data that is being read or written with GET EDIT and PUT EDIT statements and defines the conversion, if any, to be performed	17.1.3
FREE	Releases storage of a based or controlled variable	9.5.4
GET	Obtains data from an external stream file or from a character-string expression	17.1.1
GOTO	Transfers control to a labeled statement	14.6
IF	Tests an expression and establishes actions to be performed based on the result of the test	14.4
LEAVE	Transfers control out of a DO-group	14.7
ON	Establishes the action to be performed when a specified condition is signaled	15.1
OPEN	Establishes the association between a file control block and an external file	16.1.2
PROCEDURE	Specifies the point of invocation for a program, subroutine, or user-defined function	13.1.2
PUT	Transfers data to an external stream file or to a character-string variable	17.1.2
READ	Obtains a record from a file	18.1.1
RETURN	Gives control back to the procedure from which the current procedure was invoked	13.1.6
REVERT	Cancels the effect of the most recently established ON-unit	15.2
REWRITE	Replaces a record in an existing file	18.1.3
SELECT	Tests a series of expressions and establishes action to be performed based on the result of the test	14.5
SIGNAL	Causes a specific condition to be signaled	15.3
STOP	Halts the execution of the current program	14.8
WRITE	Copies data from the program to an external record file	18.1.2

7.3 Data and Variables

The statements in a PL/I program process data, generally in the form of variables that take on different values as the result of program execution. In VAX-11 PL/I, you usually must declare variables in a DECLARE statement before you can use them in other statements. Declaring a variable associates an identifier and a set of attributes with a region of storage.

Thus, when you declare a variable you must usually specify one or more data type attributes to be associated with it. Furthermore, you can specify how the variable is to be allocated by supplying a storage class attribute in the declaration. Table 7-3 is an alphabetic list of all the attributes available in VAX-11 PL/I.

Table 7-3: Summary of VAX-11 PL/I Attributes

Attribute	Use
ALIGNED	Requests alignment of bit-string variables in storage
ANY	Indicates that a parameter may have any data type
AREA	Defines a unit of storage for the allocation of based variables
{ AUTOMATIC / AUTO }	Requests dynamic allocation of storage for a variable
[BASED [(pointer-reference)]]	Indicates that a variable's storage is located by a pointer
{ BINARY / BIN }	Defines a binary base for arithmetic data
BIT	Defines bit-string data
BUILTIN	Defines a built-in function name
{ CHARACTER / CHAR } [(length)]	Defines character-string data
{ CONTROLLED / CTL }	Defines a variable whose storage is allocated and freed in successive and fixed-sequence generations
{ DECIMAL / DEC }	Defines a decimal base for arithmetic data
{ DEFINED / DEF } (variable-reference)	Indicates that a variable will share the storage allocated for another variable
dimension	Indicates that a variable is an array and defines the number and extent of its dimensions
DIRECT	Specifies that a file will be accessed only randomly
ENTRY (descriptor,...)	Describes an external procedure and its parameters
{ ENVIRONMENT / ENV } (option,...)	Specifies system-dependent information about a file
extent	Gives the length or dimension of a variable
{ EXTERNAL / EXT }	Identifies the name of a variable whose storage is referenced or defined in other procedures
FILE	Identifies a PL/I file constant or file variable

Table 7-3 (Cont.): Summary of VAX-11 PL/I Attributes

Attribute	Use
FIXED	Defines a fixed-point arithmetic variable
FLOAT	Defines a floating-point arithmetic variable
GLOBALDEF [(psect-name)]	Defines an external variable and specifies the program section in which the variable will reside
GLOBALREF	Defines an external variable whose value is defined in an external procedure
{ INITIAL / INIT } (value,...)	Provides initial values for variables
INPUT	Specifies that a file will be used for input
{ INTERNAL / INT }	Limits the scope of a variable to the block in which it is defined
KEYED	Specifies that a file may be accessed randomly by key
LABEL	Defines a label variable
length	Specifies a length for a string variable
LIKE	Copies the declaration of a structure to another structure variable
OFFSET	Defines an offset variable
OPTIONS	Specifies attribute options
OUTPUT	Specifies that a file will be used for output
parameter	Indicates that a variable will be assigned a value when the procedure is invoked
{ PICTURE / PIC } 'picture'	Specifies the format of numeric data stored in character form
{ POINTER / PTR }	Defines a pointer variable
{ POSITION / POS }	Specifies the position within a variable at which a defined variable begins
precision,[scale-factor]	Specifies the number of digits in an arithmetic variable and, with fixed-point data, the number of fractional digits
PRINT	Specifies that a file is to be formatted for printing
READONLY	Specifies that a static variable's value does not change during program execution
RECORD	Specifies that a file will be accessed by record I/O statements
REFER	Defines dynamically self-defining structures

Table 7-3 (Cont.): Summary of VAX-11 PL/I Attributes

Attribute	Use
RETURNS(returns-descriptor)	Specifies that an external entry is a function and describes the value returned by it
{ SEQUENTIAL / SEQ }	Specifies that a file may be accessed sequentially
STATIC	Requests static allocation of storage
STREAM	Specifies that a file will be accessed by stream I/O statements
UNION	Indicates that a variable will share the storage allocated for another variable
UPDATE	Specifies that records in a file may be rewritten or deleted
VALUE	Requests (1) that a global symbol be accessed by value rather than by reference, or (2) that an argument be passed to a non-PL/I procedure by immediate value
VARIABLE	Defines variable entry and file data
{ VARYING / VAR }	Defines a varying-length character string

An identifier can refer to a single variable (called a scalar variable) or to a collection of related variables. Such a collection is called an aggregate. There are two kinds of aggregate: the array, in which all members have the same data type and are referenced by relative position; and the structure, in which the members may have different data types and are referenced in a hierarchical fashion.

The following chapters provide information on these topics:

- Chapter 8 describes the data types that you can specify for variables.
- Chapter 9 describes the storage classes.
- Chapter 10 describes aggregates.
- Chapter 11 describes the DECLARE statement and the scope of a declaration.

7.4 Program Text

The text of a PL/I program consists of PL/I statements and comments. This section discusses program format, gives rules for comments, and describes the %INCLUDE statement, which allows you to include text from files or text libraries in a compilation.

7.4.1 Program Format

The source text of a PL/I program is freeform. As long as you terminate every statement with a semicolon (;), individual statements can begin in any column, spill over onto additional lines, or be written with more than one statement to a line.

Individual keywords or identifiers of a statement, however, must be confined to one line. Only a character-string constant (which must be enclosed in apostrophes) can spill over onto more than one line.

PL/I programs are easier to read and to comprehend if you follow a standard pattern in formatting. For example:

- Write source statements with no more than one statement per line.
- Use indention to show the nesting level of blocks and DO-groups.

7.4.2 Comments

A comment is an informational tool for documenting a PL/I program. To insert a comment in a program, enclose it within the character pairs /* and */. For example:

```
/* This is a comment.... */
```

Wherever the characters /* appear in a program, the compiler ignores all text until it encounters the characters */. Thus, a comment can span several lines.

The rules for entering comments are

- A comment can appear anywhere that a space can appear, that is:
  - Between any identifiers, keywords, or constants; in this context, a comment separates tokens, or discrete text items, in a statement.
  - Preceding or following punctuation marks that normally serve as delimiters, for example, spaces, tabs, or commas.
- A comment can contain any character except the pair */; comments cannot be nested.

Some examples of comments are

```
A = B + C ;            /* Add B and C */

/* *********  START OF SECOND PHASE ********* */

DECLARE/*COUNTER*/A FIXED BINARY (7);

/* This module performs the following steps:
   1. Initializes all arrays and data structures.
   2. Establishes default condition handlers.
*/
```

Although complete comments cannot be nested, you can "comment out" a statement such as

```
DECLARE EOF BIT(1); /* end-of-file */
```

This statement can be commented out by preceding the DECLARE with another /* pair. The compiler will then ignore all text, including the DECLARE statement, until it reaches the */ pair.

7.4.3 %INCLUDE Statement

An %INCLUDE statement in a PL/I source file requests inclusion of an entire file, or of a module from a library of text files. When the compiler reads the %INCLUDE statement during compilation of a source program, it begins reading from the file or module specified by %INCLUDE. When it reaches the end of the included text, it resumes reading from the previous input file.

When an %INCLUDE statement in your program requests inclusion of a module from a library, you must be sure that the PL/I compiler can find the library. Either you must specify the library explicitly in the PLI command, or request a module from one of the libraries that the compiler searches by default.

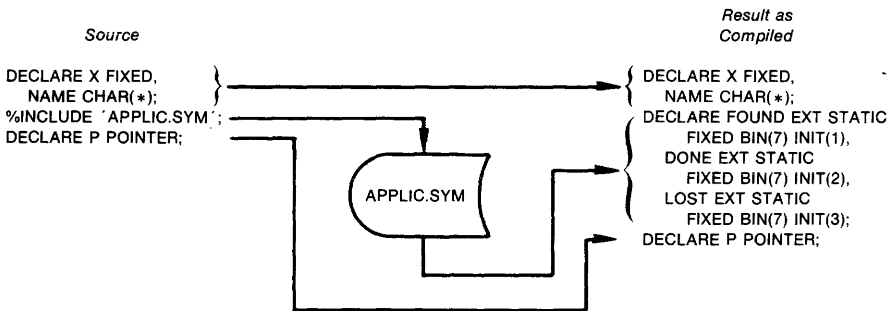


Figure 7-1: Using the %INCLUDE Statement

Included text can contain an %INCLUDE statement. The maximum depth to which included text can be nested is four.

The %INCLUDE statement format is

```
%INCLUDE { 'file-spec'        } ;
         { text-module-name   }
```

file-spec

A file specification enclosed in apostrophes. The default file type is PLI. The entire file is included in the compilation.

text-module-name

The 1- to 31-character name of a text module in a library of INCLUDE files or other text modules. Only the contents of the module are included in the compilation.

For example, the following specifications are different:

```
%INCLUDE 'STATE';
%INCLUDE STATE;
```

In the first example, PL/I assumes that STATE is a file specification and looks for the file STATE.PLI in the directory that contains the file being compiled. In the second example, PL/I searches any library files specified in the PLI command for a module named STATE.

When you use the %INCLUDE statement to request inclusion of a text module from a library file, you must ensure that the compiler can locate the appropriate library. If PL/I cannot locate a specified file or module, it issues a fatal error message and terminates the compilation.

Chapter 8

Data Types

This chapter includes the following topics:

- A brief summary of the data types
- Arithmetic data types, which are used to represent numeric values
- Character-string data, which consists of sequences of ASCII characters
- Bit-string data, which consists of sequences of binary digits (bits)

8.1 Summary of Data Types

VAX-11 PL/I supports the following computational data types:

- The arithmetic data types define values that can be used in arithmetic computation. They are

  - fixed-point (binary and decimal integers and fractions).
  - floating-point (binary and decimal).
  - pictured (fixed-point data stored in character form).

  Sections 8.2.1 through 8.2.4 describe them.

- Character-string data consists of a sequence of ASCII characters. (See Section 8.3.)
- Bit-string data consists of sequences of binary digits. (See Section 8.4.)

The data types listed below represent noncomputational program values that are used within a PL/I program for control. Each of them is described along with its associated function in the indicated section.

- Entry constants and variables are used to invoke procedures through specified entry points. (See Section 13.2.1.)
- Label variables and constants provide you with a flexible means of control within a program. (Section 14.6 contains a description of label data.)
- File variables and constants provide access to files. (See Section 16.1.1.)
- Pointers represent the location in memory of data, and are used to access based variables in areas and data in system-allocated buffers. (See Section 9.5.1.1.)
- Areas are regions of storage in which based variables may be allocated and freed. Offsets represent the location of a based variable in an area. (See Section 9.5.1.2.)

8.2 Arithmetic Data Types

Arithmetic data types are used for variables on which arithmetic calculations are to be performed. The arithmetic data types supported by VAX-11 PL/I are

- Fixed-point—for binary and decimal data with a fixed number of fractional digits.
- Floating-point—for calculations on very large or very small numbers, with the decimal point (number of fractional digits) allowed to "float."
- Pictured—for fixed-point decimal data that is stored internally in character form, with special formatting characters.

Sections 8.2.1 through 8.2.4 describe these four data types. Section 8.2.5 describes the precision of all arithmetic data types.

When you declare an arithmetic variable, you do not always have to define all its characteristics, or attributes; the PL/I compiler makes assumptions about attributes that are not explicitly defined. For example:

```
DECLARE NUMBER FIXED;
```

The FIXED attribute implies the attributes BINARY(31,0). Thus, the variable NUMBER has the attributes FIXED BINARY(31,0).

Table 8–1 shows the default attributes implied by each arithmetic data attribute.

Table 8–1: Implied Attributes for Computational Data

Specified	Implied
FIXED	BINARY(31,0)
FLOAT	BINARY(24)
BINARY	FIXED(31,0)
DECIMAL	FIXED(10,0)
FIXED BINARY	(31,0)
FLOAT BINARY	(24)
FIXED DECIMAL	(10,0)
FIXED DECIMAL(p)	(p,0)
FLOAT DECIMAL	(7)

8.2.1 Fixed-Point Binary Data

The attributes FIXED BINARY are used to declare integer variables in PL/I. The BINARY attribute is implied by FIXED. The declaration of a single fixed-point binary variable is of the form

```
DECLARE identifier FIXED [ BINARY ] [(p[,q])];
                         [ BIN    ]
```

identifier

The name used to refer to the variable.

p

An integer constant giving the total number of binary digits used to represent values of the variable. The value must be in the range

$1 \leq p \leq 31$

q

An integer constant giving the number of fractional binary digits in values of the variable. The value must be in the range

$-31 \leq q \leq p$

If you omit p and q, the default values are p=31, q=0.

Precision is an integer from 1 to 31. If you do not specify the precision, the default is 31. The precision that you specify establishes the storage allocated for the variable and the range of values that the variable can take on, as follows:

Precision	Storage	Maximum Range
$1 \leq p \leq 7$	byte (8 bits)	-128 through 127
$8 \leq p \leq 15$	word (16 bits)	-32,768 through 32,767
$16 \leq p \leq 31$	longword (32 bits)	-2,147,483,648 through 2,147,483,647

There is no representation in VAX-11 PL/I for a fixed-point binary constant. Instead, integer constants are represented as fixed decimal. However, fixed decimal integer constants (and variables) are converted to fixed binary when combined with fixed binary variables in expressions. For example, assume that I is a fixed binary variable:

```
I = I+3;
```

In this example, the integer 3 is represented as fixed decimal, but PL/I converts it to fixed binary when evaluating the expression.

Because fixed binary variables have a maximum precision of 31, fixed binary integers can have values only in the range of -2,147,483,648 through 2,147,483,647. An attempt to calculate a binary integer outside this range, in a context that requires an integer value, signals the FIXEDOVERFLOW condition.

8.2.2 Fixed-Point Decimal Data

Fixed-point decimal data is used in calculations where exact decimal values must be maintained, for example, in financial applications. Fixed-point decimal data with a scale factor of zero may also be used whenever integer data is required.

This section is divided into the following parts:

- Constants
- Variables
- Use in expressions

8.2.2.1 Fixed-Point Decimal Constants

A fixed-point decimal constant can have between 1 and 31 decimal digits (0 through 9) with, optionally, a decimal point and/or a sign. If there is no

decimal point, PL/I assumes it to be immediately to the right of the right-most digit. Some examples of fixed-point decimal constants are

```
12
4.56
12345.54
-2
.0004
01.
```

The precision (p) of a fixed-point decimal value is the total number of digits in the value. The scale factor (q) is the number of digits to the right of the decimal point, if any. The scale factor cannot be greater than the precision.

8.2.2.2 Fixed-Point Decimal Variables

The attributes FIXED DECIMAL are used to declare fixed-point decimal variables. The FIXED attribute is implied by DECIMAL. The form of a declaration of a single fixed-point decimal variable is

DECLARE identifier [FIXED] { DECIMAL | DEC } [(p[,q])];

identifier

The name to be used for the variable.

p

An integer constant giving the total number of decimal digits used to represent values of the variable. The value must be in the range

$1 \le p \le 31$

q

An integer constant giving the number of fractional digits in values of the variable. The value must be in the range

$0 \le q \le p$

If you omit p and q, the default values are p=10, q=0.

Some examples of fixed-point decimal declarations are

```
DECLARE PERCENTAGE FIXED DECIMAL (5,2);
DECLARE TONNAGE FIXED DECIMAL (9);
```

8.2.2.3 Using Fixed-Point Data in Expressions

You cannot use fixed-point decimal data with a nonzero scale factor in calculations with binary integer variables. If you must combine the two types of data, use the DECIMAL built-in function (described in Section

19.2) to convert the binary value to a scaled decimal value before attempting an arithmetic operation. For example:

```
DECLARE I FIXED BINARY,
   SUM FIXED DECIMAL (10,2);

SUM = SUM + DECIMAL (I);
```

8.2.3 Floating-Point Data

The floating-point data types provide a way to express very large and very small numbers, for example, in scientific calculations. All floating-point calculations are performed on values in one of the VAX–11 binary floating-point formats. In general, the precision of the result is determined by the maximum precision of any operands in the operation. The numerical result of an operation is rounded to the result precision; therefore, the results of most operations are approximate.

This section is divided into the following parts:

- Constants
- Variables
- Use in expressions
- Floating-point data formats

8.2.3.1 Constants

A floating-point constant can have one or more of the decimal digits 0 through 9 (with an optional decimal point), followed by the letter E and from one to five decimal digits representing a power of 10. The floating-point value and the integer exponent can both be signed. The first portion of the value, to the left of the letter E, is called the mantissa. The value to the right of the letter E is called the exponent.

Some examples of floating-point constants are

```
2E10
-3E8
32E-8
.45632E16
```

The decimal precision of each of these values is the number of digits in the mantissa.

If you write a constant without the E and the exponent, it is considered to be fixed-point decimal. However, you can use such constants freely in expressions involving floating-point data.

8.2.3.2 Variables

The keyword FLOAT identifies a floating-point variable in a declaration. The form of the declaration of a single floating-point variable is

```
DECLARE identifier FLOAT [ BINARY  ] [(p)];
                         [ DECIMAL ]
```

identifier

The name to be used for the variable.

[BINARY | DECIMAL]

An attribute that determines whether the variable is to be floating-point binary or decimal. The primary effect of this attribute is to determine how the compiler interprets the precision attribute. BINARY is the default.

p

An integer constant that specifies the precision of the variable. For a floating-point binary variable, p is the number of bits to be maintained in the mantissa; the range of p is 1 to 113, and the default value is 24. For a floating-point decimal variable, p is the number of decimal digits to be maintained in the mantissa; the range of p is 1 to 34, and the default value is 7. Section 8.2.3.4 describes the effect of different values of p on floating-point variable representation.

Note that you can use either BINARY or DECIMAL to declare a floating-point value. Since the internal representation of floating-point variables is binary, it is recommended that you use BINARY FLOAT to declare variables (this is the default). In any event, you should declare all floating-point variables using the same base.

8.2.3.3 Using Floating-Point Data in Expressions

You can use both integer and scaled decimal constants freely in floating-point expressions. An arithmetic constant is always converted to the appropriate internal representation for use in a floating-point operation. The target type for the conversion depends on the context. In the following example

```
DECLARE X FLOAT BINARY (53);
X = X + 1.3;
```

the constant 1.3 is converted to floating point when the expression is evaluated.

8.2.3.4 G_FLOAT and H_FLOAT Support

If your main program or a procedure that it invokes uses floating-point variables with a binary precision greater than 53 (decimal precision greater than 15), and your computer does not include hardware support for G- and H-floating data, you must provide your program with access to a software emulation routine for these types. Include the following lines in your MAIN procedure:

```
DECLARE SS$_OPCDEC FIXED BIN GLOBALREF VALUE;
  ON VAXCONDITION(SS$_OPCDEC) CALL RESIGNAL();
```

Compile the program with the /G_FLOAT qualifier, and link it as follows:

```
$ LINK your_prog,SYS$LIBRARY:STARLET/INCLUDE=LIB$ESTEMU
```

None of this is necessary for external procedures not having OPTIONS(MAIN).

8.2.3.5 Floating-Point Data Formats

VAX-11 PL/I supports four types of floating-point values. Table 8-2 summarizes the ranges of precision for each type.

Table 8-2: VAX-11 Floating-Point Types

Floating-Point Type[1]	Sign Bits	Exponent Bits	Fractional Bits
F (single precision)	1	8	24
D (double precision)	1	8	53
G	1	11	53
H	1	15	113

1. Types G and H require a VAX-11 hardware option; types F and D are available on all VAX processors.

The approximate ranges of the VAX-11 floating-point formats are as follows:

- F format: $0.29*10^{-38}$ to $1.7*10^{38}$
- D format: same as F, but with more precise mantissa (see Table 8-2)
- G format: $0.56*10^{-308}$ to $0.9*10^{308}$
- H format: $0.84*10^{-4932}$ to $0.59*10^{4932}$

The PL/I compiler selects the appropriate VAX-11 floating-point type based on, first, the precision you specify and, second, a compile-time qualifier on the PLI command. The types are selected as shown in Table 8-3. The default is F (single precision).

Table 8-3: Floating-Point Types Used by PL/I

Range of p (DECIMAL)	Range of p (BINARY)	Floating-point Type
$1 \le p \le 7$	$1 \le p \le 24$	F
$8 \le p \le 15$	$25 \le p \le 53$	D or G[1]
$16 \le p \le 34$	$54 \le p \le 113$	H[2]

1. D is used unless /G_FLOAT is requested at compile time.
2. H is possible only if /G_FLOAT is requested at compile time.

8.2.4 Pictured Data

Use pictured data when you want to manipulate a quantity arithmetically and accept or display its value using a special format. Pictured variables are especially useful in applications that require values to be shown with special symbols, such as commas, dollar signs, or debit indicators (DB).

This section describes

- Pictured variables—variables declared with the PICTURE data attribute.
- Assigning values to pictured data—the process by which a value is assigned to a pictured variable or written out with the P format item.
- Extracting values from pictured data—the process by which a pictured value is assigned to other variables or acquired with the P format item.
- Picture characters—the special characters that make up a specification in the PICTURE attribute and in the P format item.

Although the formatting possible with pictured data is useful in many applications, pictured data is much less efficient than fixed-point decimal data in computations. Therefore, do not use pictured data unless you need the formatting.

8.2.4.1 Pictured Variables

A pictured variable has the attributes of a fixed-point decimal variable, but values assigned to it are stored internally as character strings. Such a character string contains digits representing the variable's numeric value as well as such special symbols as the dollar sign. When the value of a pictured variable is written out by, for example, the PUT LIST statement, the internally stored character string is placed in the output stream. The value that appears on a line printer or terminal thus contains a fixed-point decimal number that has been "edited" with the requested special symbols.

The declaration of a single pictured variable has the form

DECLARE identifier { PICTURE | PIC } 'picture' ;

identifier

The name to be used for the variable.

'picture'

A string of picture characters that define the representation of the variable. (These characters and their uses are described in Section 8.2.4.4.) The apostrophes surrounding the picture are required syntax.

A picture specification (or picture) describes both the numeric attributes of a pictured variable and its output format. A simple picture looks like this in a DECLARE statement:

```
DECLARE CREDIT PICTURE '$99999V.99DB';
```

This statement declares the variable CREDIT as a pictured variable; its picture comprises the characters within the apostrophes.

The two assignments

```
CREDIT = 12443.00;
```

and

```
CREDIT = -12443.00;
```

would look like this on output:

```
$12443.00                /* a positive value (credit) */

$12443.00DB              /* a negative value (debit) */
```

8.2.4.2 Assigning Values to Pictured Variables

Assignment of a computational value to a pictured variable is performed in the following two steps:

1. The value is converted to fixed decimal, with precision and scale as specified by the picture.
2. The resulting fixed decimal value is edited into the pictured variable.

If PL/I cannot perform one of these steps in a meaningful fashion, an error occurs. The following examples show two programming errors that are common in assignments to pictured variables.

```
CREDIT = '$12443.00';
```

This example signals the ERROR condition, because the character string contains a dollar sign and is therefore not convertible to fixed-point decimal. The value assigned to CREDIT should be either '12443.00' or simply 12443.00, both of which result in the same value assigned to CREDIT.

If a negative value is assigned to a pictured variable, the picture must include one of the sign picture characters (such as DB). If, for example, the picture of CREDIT did not contain the DB characters, then the assignment

```
CREDIT = -12443.00;
```

would signal the FIXEDOVERFLOW condition, because the sign would be lost.

In some circumstances (for example, with the READ statement), it is possible to assign a value to a pictured variable that is not valid with respect to the variable's picture specification. In such cases, the VALID built-in function (described in Section 19.2) can be used to validate the contents of the variable.

8.2.4.3 Extracting Values from Pictured Data

When you use a pictured value in an arithmetic context (such as assignment to an arithmetic variable), the picture is used to extract the fixed-point decimal number from the character string that internally represents the pictured value. Extraction also occurs when you input a pictured value with the GET EDIT statement and the P format item. If the contents of the pictured variable or input item do not conform to the picture, an error occurs.

For example, in the picture for CREDIT

```
DECLARE CREDIT PICTURE '$99999V.99DB';
```

the 9 character specifies the position of a decimal digit; since the picture contains seven of these, the fixed-point decimal precision of CREDIT is 7. The V character separates the integral and fractional digits; since there are two 9 characters to the right of the V, the scale factor of CREDIT is 2. The V character is unique among picture characters in that it specifies only a numeric property; it does not cause a decimal point (or any other character) to appear in the internal representation of CREDIT. Therefore, a period picture character (.) can be included after the V to ensure that the output value has a decimal point in the correct place.

The period and dollar sign are always inserted in the internal representation and the output value regardless of CREDIT's numeric value.

The picture character DB appears only when the value of CREDIT is less than zero; otherwise, two spaces appear in the indicated positions. The DB character also indicates that a value of CREDIT is numerically negative, so that if CREDIT is later assigned to an arithmetic variable, the variable will be given a negative value.

8.2.4.4 Picture Characters

An individual picture character, and its position in the picture, indicate the interpretation of an associated position in the pictured value. Table 8-4 lists the characters that can appear in a picture.

Table 8-4: Picture Characters

Character	Meaning
9	Decimal digit, including leading zeros
Z	Decimal digit with leading-zero suppression
*	Decimal digit with asterisk for leading zero
Y	Decimal digit with space for any zero
V	Position of assumed decimal point
(n)	Iteration factor for subsequent character
T	Position of digit and encoded plus sign or minus sign
I	Position of digit and encoded plus sign if number ≥ 0
R	Position of digit and encoded minus sign if number < 0
.	Position at which decimal point is inserted
,	Position at which comma is inserted
/	Position at which slash is inserted
B	Position at which space is inserted
$	Position[s] of [drifting] dollar sign
+	Position[s] of [drifting] plus sign if number ≥ 0
-	Position[s] of [drifting] minus sign if number < 0
S	Position[s] of [drifting] plus sign or minus sign
CR	Positions at which 'CR' is inserted if number < 0
DB	Positions at which 'DB' is inserted if number < 0

Although all picture characters are shown here in uppercase form, the lowercase equivalents function identically.

Any picture character that can appear more than once in a picture may be preceded by an iteration factor, which must be a positive integer constant enclosed in parentheses. For example, the picture

```
'(4)9'
```

is the same as

```
'9999'
```

The following paragraphs describe the picture characters in more detail.

Decimal Place Character (V)

The V character shows the position of the "assumed" decimal point, or, in other words, the scale factor for the fixed-point decimal value. It does not cause a decimal point to appear. (Use the period insertion character for this purpose.) The following rules apply to the V character:

- Only one V character may appear in a picture.
- If a picture does not contain the V character, it is assumed to be at the right end of the picture.
- If a fixed-point value assigned to a pictured variable has fewer integral digits than are indicated by the picture characters to the left of the V, then the integral value of the pictured variable is extended on the left with zeros. If the assigned value has too many integral digits, the value of the pictured variable is undefined and the FIXEDOVERFLOW condition is signaled.
- If a fixed-point value assigned to a pictured variable has fewer fractional digits than are indicated in the picture, then the fractional value of the pictured variable is extended on the right with zeros. If the assigned value has too many fractional digits, then the excess fractional digits are truncated on the right; no condition is signaled. Thus, if the V character is the last character in the picture or is omitted, assigned fixed-point values are truncated to integers.

The following example illustrates the effect of the V character:

```
DECLARE PRICE PICTURE '$$9V.99',
    BAD_PRICE PICTURE '$$9.99';
        PRICE = .98;       /* Output as $0.98 */
        BAD_PRICE = .98;   /* Output as $0.00 */
        PRICE = 98;        /* Output as $98.00 */
        BAD_PRICE = 98;    /* Output as $0.98 */
```

In this example, note that the variable PRICE, which contains the V character, represents the value properly. The variable BAD_PRICE, which contains only the period insertion character, has an assumed V character at the end of the picture, which causes the variable to misrepresent the value.

Digit Characters (9, Z, *, Y)

All of these characters mark the positions occupied by decimal digits. The number of these characters present in a picture specifies the number of digits, or precision, of the fixed-point decimal value of the pictured variable.

- The position occupied by 9 always contains a decimal digit, whether or not the digit is significant in the numeric interpretation of the pictured value. Thus, leading zeros at positions occupied by a 9 are output.

- The position occupied by Z contains a decimal digit only if the digit is significant in the integral portion of the numeric interpretation; if the digit is an insignificant, or leading, zero, it is replaced by a space.
  - The Z character must not appear in the same picture with the character *. It must not appear to the right of the characters 9, T, I, or R nor to the right of a drifting string (see "Drifting Characters" below).
  - If the Z character appears to the right of the V character, then all digits to the right of the V must be indicated by Z characters. Fractional zeros are then suppressed only if all fractional digits are zero and all of the integral digits are suppressed; in that case, the internal representation contains only spaces in the digit positions.
- The position occupied by the * character functions identically with the Z character, except that leading zeros are replaced by asterisks instead of spaces.
- The position occupied by the Y character contains a decimal digit only if the digit is not zero. All zeros in the indicated positions, whether significant or not, are replaced by spaces.

Encoded-Sign Characters (T, I, R)

The characters T, I, and R are digit characters that may be used wherever 9 is valid. Each represents a digit that has the sign of the pictured value encoded in the same position. Only one can be used in a picture.

An encoded-sign character cannot be used in a picture that contains an S, +, -, CR, or DB (described below).

The meanings of the characters are as follows:

- The T character indicates that the position contains an encoded minus sign if the numeric value is less than zero and an encoded plus sign if the numeric value is greater than or equal to zero. These encoded-sign digits are represented internally and in output by the ASCII characters shown in Table 8-5.
- The I character indicates an encoded plus sign if the numeric value is greater than or equal to zero. Otherwise, the position contains an ordinary digit.
- The R character indicates an encoded minus sign if the numeric value is less than zero. Otherwise, the position contains an ordinary digit.

Table 8-5 shows the ASCII characters that indicate digits with encoded signs: +digit means the digit with an encoded plus sign; -digit, an encoded minus sign. The characters in Table 8-5 are used in the internal representation of a pictured value and must be used for input of an encoded-sign digit from a stream file.

Table 8-5: ASCII Representation of Encoded-Sign Digits

Digit	ASCII Character	Digit	ASCII Character
+0	{	-0	}
+1	A	-1	J
+2	B	-2	K
+3	C	-3	L
+4	D	-4	M
+5	E	-5	N
+6	F	-6	O
+7	G	-7	P
+8	H	-8	Q
+9	I	-9	R

Drifting Characters ($, +, , S)

The drifting characters can be used to indicate digits, and they also indicate a symbol to be inserted when, for example, a pictured value is written out by PUT LIST.

- The dollar sign ($) causes a dollar sign to be inserted.
- The plus sign (+) causes a plus sign to be inserted if the numeric value is greater than or equal to zero.
- The minus sign (-) causes a minus sign to be inserted if the numeric value is less than zero.
- The S character causes a plus sign to be inserted if the numeric value is greater than or equal to zero, and a minus sign if the value is less than zero.

If one of these characters is used alone in the picture, it marks the position at which a special symbol or space is always inserted, and it has no effect on the value's numeric interpretation. In this case, the character must appear either before or after all characters that specify digit positions.

However, if a series of n of these characters appears, then the rightmost n–1 of the characters in the series also specify digit positions. If the digit is a leading zero, the leading zero is suppressed, and the leftmost character "drifts" to the right; the character appears either in the position of the last drifting character in the series or immediately to the left of the first significant digit, whichever comes first. Used this way, the n–1 drifting characters also define part of the numeric precision of the pictured variable, since they describe at least some of the positions occupied by decimal digits. For

an example of this behavior by a drifting character (the dollar sign), see the decimal place character (V) above.

The following additional rules apply to drifting characters:

- A drifting string is a series of more than one of the same drifting character. Only one drifting string can appear in the picture; any other drifting characters can be used only singly and therefore designate insertion characters, not digits.
- The characters Z and * cannot appear to the right of a drifting string.
- A digit position cannot be specified (for instance, with a 9) to the left of a drifting string.
- A drifting string can contain the V character and one of the insertion characters (defined below). The following additional rules apply:
    - If the drifting string contains an insertion character, it is inserted in the internal representation only if a significant digit appears to its left. In the position of the insertion character, a space appears if the the leftmost significant digit is more than one position to the right; the drifting symbol appears if the next position to the right contains the leftmost significant digit.
    - If the drifting string contains a V character, all digit positions to the right of the V (the fractional digits) must also be part of the drifting string. In this case, insignificant fractional digits are suppressed only when all integral and fractional digits are zeros: they are replaced by spaces in the internal representation. If any digit is not zero, all fractional digits appear as actual digits.
    - Any insertion characters immediately to the right of a drifting string are considered part of it.

Insertion Characters

The insertion characters indicate that characters are inserted between digits in the pictured value. The insertion characters are the comma (,), period (.), slash (/), and the space (B). The B character indicates that a space is always inserted at the indicated position.

The drifting characters also function as insertion characters when used singly (that is, not as part of a drifting string).

Note that the period (.) does not imply a decimal place character (V). This is illustrated by the example of the decimal place character, above.

The following rules describe insertion by the comma, period, and slash insertion characters.

- If zero suppression occurs, the insertion character is inserted only in these cases:
  - A significant digit appears immediately to its left.
  - The V character appears immediately to its left, and the fractional part of the numeric value contains significant digits.
- To guarantee that the decimal point is in the same position in both the numeric and character interpretations, the V and period characters must be immediately adjacent. Note, however, that if the period precedes the V, then it is suppressed if there are no significant integral digits, even though all the fractional digits are significant. This property can make fractions appear to be integers when the internal (character) value is displayed. Consequently, the period should immediately follow the V character; it will then be in the correct location and will appear whenever any fractional digit is significant. The following example illustrates correct and incorrect placement of the period:

```
DECLARE NUM PICTURE 'ZZZV.ZZ',
    BAD_NUM PICTURE 'ZZZ.VZZ';
NUM=0.02;          /* Output as .02 */
BAD_NUM=0.02;      /* Output as 02 */
```

- Other insertion characters, such as the comma, can be used to separate the integral and fractional portions of a number. However, the comma should not be used with GET LIST input, because in that context it separates different data items in the input stream.

Credit (CR) and Debit (DB) Characters

These picture characters are always specified as the character pairs CR and DB. If either pair is included, it appears if the numeric value is less than zero. In each case, the associated positions contain two spaces if the numeric value is greater than or equal to zero.

The characters are always inserted with the same case as used in the picture. If the lowercase form cr is used in the picture, lowercase letters are inserted in the pictured value; if the combination Cr is used, then Cr is inserted.

The credit and debit characters cannot be combined in one picture, nor can they be used in the same picture as any other character that specifies the sign of the value (S, +, and –). In addition, they must appear to the right of all picture characters specifying digits.

8.2.5 Precision of Arithmetic Data Types

The precision attribute applies to binary and decimal data; the precision of an item is the total number of decimal or binary digits used to represent a

value. You can specify the precision and scale of an arithmetic variable in a DECLARE statement in any of the following formats, depending on the numeric base of the data item:

```
BINARY [ FIXED ] [ (precision[,scale-factor]) ]
[ BINARY ] FLOAT [ (precision) ]
DECIMAL [ FIXED ] [ (precision[,scale-factor]) ]
DECIMAL FLOAT [ (precision) ]
```

The precision of a floating-point data item is the number of decimal or binary digits in the mantissa of the floating-point representation.

The ranges of values you can specify for the precision for each arithmetic data type, and the defaults applied if you do not specify a precision, are summarized as follows:

Data Type Attributes	Precision	Scale Factor	Default Precision
BINARY FIXED	$1 \le p \le 31$	$\le p$	31
BINARY FLOAT	$1 \le p \le 113$	-	24
DECIMAL FIXED	$1 \le p \le 31$	$\le p$	10
DECIMAL FLOAT	$1 \le p \le 34$	-	7

If no scale factor is specified for fixed-point data, the default is zero.

8.2.6 Scale of Fixed-Point Data Types

In addition to the precision attribute, fixed-point data may also have the scale attribute, which is the number of fractional bits or digits contained within the specified precision. The scale factor q specifies that all values of the fixed-point variable are "scaled" by the factor 2^{-9} for binary data or 10^{-9} for decimal data.

Data Type Attributes	Scale Factor	Default Scale
BINARY FIXED	$-31 \le p \le 31$	0
DECIMAL FIXED	$0 \le p \le 31$	0

For example:

```
DECLARE x FIXED DECIMAL(10,3);
```

indicates that the value of x has 10 decimal digits, but 3 of those are fractional. In effect, this is similar to multiplying or dividing the decimal number by a factor of 10.

Positive scale factors for fixed binary numbers function the same as scale factors for fixed decimal numbers. A negative scale factor indicates the number of fractional bits that are shifted from the left to the right. For a fixed-point binary number, the scale factor has the effect of multiplying or dividing the number by a factor of 2.

Even though arithmetic operands can be of different arithmetic types, all operations must actually be performed on objects of the same type. Consequently, the compiler may convert operands to a derived type. Therefore, when you declare a fixed binary number with a scale factor, and assign it a decimal value, the results may not be what you expect. This is because the binary scale factor left-shifts the specified number of bits to the right of the decimal point. During conversion to a decimal representation, the difference between the resulting binary number and its decimal representation is not the equivalent of dividing or multiplying the decimal number by 10. Instead, the binary number is divided or multiplied by 2 and then converted to its decimal representation.

In addition, excess fractional digits may be truncated, and no condition is signaled. Any resulting loss of precision may be difficult to detect because truncated fractional digits do not signal a condition.

8.3 Character-String Data

A character string is a sequence of zero or more ASCII characters (see Appendix C for a table of the ASCII characters). A character-string value can consist of any ASCII characters, to a maximum length of 32767 characters.

This section is divided into the following parts:

- Constants
- Variables

8.3.1 Character-String Constants

When you use character-string constants in a program, you must enclose the strings in apostrophes, as shown in the following examples:

```
'Total is:'
'Enter your name and age'
'Error - value is out of range'
```

To specify a string containing a literal apostrophe, use two apostrophes within the string, for example:

```
'Life isn''t fair'
```

When a character string that has embedded apostrophes is specified as shown above, the final result contains only a single apostrophe.

Note that the quotation mark (") is not a legal delimiter for PL/I character constants.

8.3.2 Character-String Variables

The CHARACTER keyword identifies a character-string variable in a declaration. The addition of the VARYING keyword indicates a varying character-string variable. The format for specifying a character-string variable is

```
                      { CHARACTER }        [ VARYING ]
DECLARE identifier    {           }  [(n)] [         ] ;
                      { CHAR      }        [ VAR     ]
```

identifier

The name to be used for the variable.

n

Specifies the length of the variable, that is, the number of bytes needed to contain its value. The maximum value for n is 32767. The length attribute specifies either the length of all values of the variable (fixed-length strings) or the maximum length of a value of the variable (varying-length strings). If n is not specified, PL/I uses the default length of one character, or byte. The rules for specifying n are as follows:

- For a static variable declaration, n must be an integer constant.
- In the declaration of a parameter or in a parameter or returns descriptor, n may be specified as an integer constant or as an asterisk (*). The resulting string is fixed length unless VARYING is also specified.
- For an automatic, based, or defined variable, n may be specified as an integer constant or as an expression. In the case of automatic or defined variables, the expression must not contain any variables or functions that are declared in the same block, except for parameters.

If specified, n must immediately follow the keyword CHARACTER and must be enclosed in parentheses.

[VARYING]
[VAR]

Indicates a varying-length character-string variable. The effect of this attribute is described below.

8.3.2.1 Fixed-Length Character-String Variables

For a particular allocation of a fixed-length character-string variable, all its values have the same length. When a program assigns a value to a fixed-length character-string variable, however, the value is not always exactly

the same as the length defined for the variable. Depending on the size of the value, PL/I does the following:

- If the value is smaller than the length of the character string, PL/I pads the value with spaces on the right. For example:

  ```
  DECLARE STRING CHARACTER (10);
  STRING = 'ABCDEF';
  ```

  The final value of the variable STRING is ´ABCDEF   ´, that is, the characters ABCDEF followed by four space characters.

- If the value is larger than the length of the variable, PL/I truncates the value on the right. For example:

  ```
  DECLARE STRING CHARACTER (4);
  STRING = 'ABCDEF';
  ```

  Here, the final value of STRING is ´ABCD´, that is, the first four characters of the value ´ABCDEF´.

8.3.2.2 Varying-Length Character-String Variables

In a varying character-string variable, the length is not fixed. The length specified in the declaration of the variable defines the maximum length of any value that can be assigned to the variable. Each time a value is assigned, the current length changes. For example:

```
DECLARE NAME CHARACTER (20) VARYING;
NAME = 'COOPER';
NAME = 'RANDOM FACTOR';
```

The declaration of the variable NAME indicates that the maximum length of any character-string value it can have is 20. The current length becomes 6 when NAME is assigned the value ´COOPER´; the length becomes 13 when NAME is assigned the value ´RANDOM FACTOR´; and so on.

When a varying character string is assigned a value with a length greater than the maximum defined, the value is truncated on the right.

The initial length of an automatic varying-length character-string variable is undefined unless the variable is initialized.

You can use the LENGTH built-in function (described in Section 19.2) to determine the current length of any string.

8.4 Bit-String Data

A bit string consists of a sequence of binary digits, or bits. It may be used as a Boolean value, which has one of two states: true (if any bit is 1) or false (if all bits are 0).

Like a fixed-length character string, a bit string has a fixed length defined in the declaration or specified by the number of bits in a bit-string constant. The maximum length of any bit string is 32767 bits. Bit-string variables cannot be declared with the VARYING attribute.

Sophisticated applications that depend on the internal representation of bit strings and other types of data may not be directly transportable from other PL/I implementations to VAX-11 PL/I. In VAX-11, bit strings are stored in memory with the leftmost bit (as represented by PUT LIST) in the lowest memory location, and bits following the leftmost in successively higher memory locations. This representation of a bit string by PUT LIST is reversed with respect to a conventional picture of memory locations, in which higher locations appear on the left, not on the right. For example:

```
DECLARE ABIT BIT (10);
ABIT = '1011'B;
```

A memory diagram of the storage resulting from this assignment would look like this:

```
      ... 0  0  0  0  0  0  1  1  0  1 ...
      HIGH MEMORY     LOW MEMORY
   <- LOCATIONS       LOCATIONS ->
```

All this is of no concern until you try to interpret non-bit-string data as a bit string. For example, a fixed binary value is stored with the sign bit in the highest memory location, the most significant bit in the next highest location, and so on to the least significant bit in the lowest memory location. Thus, a FIXED BINARY (7) variable with a value of 2 would appear in memory as follows:

```
      ... 0  0  0  0  0  0  1  0 ...
      HIGH MEMORY     LOW MEMORY
   <- LOCATIONS       LOCATIONS ->
```

Should you treat this storage as a bit string (for example, by using it as the argument of the UNSPEC built-in function in a PUT LIST statement), the result would be

```
'01000000'B
```

If you are accustomed to using PL/I on computers other than VAX-11, this result may not be what you expect.

Consult the *VAX Architecture Handbook* for detailed information about the VAX-11 representation of data. The *VAX-11 PL/I Encyclopedic Reference* provides extensive information about the internal representation of PL/I data types.

The remainder of this section is divided into the following parts:

- Constants
- Variables
- Alignment
- The use of bit strings to represent integers

8.4.1 Bit-String Constants

To specify a bit-string constant, enclose the string in apostrophes and follow the closing apostrophe with the letter B. Some examples are

```
'0101'B
'10101010'B
'1'B
```

The length of a bit-string constant is always the number of binary digits specified; the B does not count in the length of the string. A bit-string constant can be specified with a maximum of 1000 characters between the apostrophes.

You can also specify a bit-string constant using the syntax

```
'character-string'Bn
```

where n is the number of bits to be represented by each character in the string. This format allows you to specify bit-string constants with bases other than 2. For example:

```
'EF8'B4
'117'B3
'223'B2
```

These constants specify the hexadecimal value EF8, the octal value 117, and the base 4 value 223. All such constants are stored internally as bit strings, not as integer representations of the value.

The valid characters for each type of bit-string constant are listed below:

- For B or B1, only the characters 0 and 1 are valid.
- For B2, only the characters 0, 1, 2, and 3 are valid.
- For B3, only the characters 0, 1, 2, 3, 4, 5, 6, and 7 are valid.
- For B4, the characters 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, and F are valid. (The letters A through F can be either upper- or lowercase.)

Using the B format items, you can also acquire or output (with the GET EDIT and PUT EDIT statements) bit-string data in binary, base 4, octal, or hexadecimal format. These format items are described in Section 17.3.1.

8.4.2 Bit-String Variables

Use the BIT attribute to declare a bit-string variable. The form of the declaration of a single bit-string variable is

```
DECLARE identifier BIT [(n)] [ALIGNED];
```

identifier

The name to be used for the variable.

n

Specifies the length of the variable. The range is 0 to 32767; the default length is one bit. The rules for specifying n are as follows:

- If BIT is specified for a static variable declaration or in a returns descriptor, length must be an integer constant.
- If BIT is specified in the declaration of a parameter or in a parameter descriptor, length may be specified as an integer constant or as an asterisk (*).
- If BIT is specified for an automatic, based, or defined variable, length may be specified as an integer constant or as an expression. In the case of automatic or defined variables, the expression must not contain any variables or functions that are declared in the same block, except for parameters.

If specified, the length must immediately follow the keyword BIT.

ALIGNED

Specifies that the bit-string variable must be aligned on a byte boundary. Section 8.4.3 describes the uses and implications of this attribute.

A program can assign to a bit-string variable a value larger or smaller than the variable's defined length. In such cases, PL/I does the following:

- If the assigned string is shorter than the defined length, PL/I pads the bit-string value with zeros in the direction of least significance. The "less significant" bits are on the right, as the string is represented by PUT LIST.
- If the assigned string is longer, PL/I truncates the least significant bits from the bit-string value.

You can convert bit-string variables to other data types; however, there are some precautions you must observe if you do so. Section 8.4.4 provides details.

8.4.3 Alignment of Bit-String Data

PL/I distinguishes between aligned and unaligned bit-string variables. (Bit-string constants are always unaligned.) A bit-string variable is aligned only if it is declared with the ALIGNED attribute, as shown in this example:

```
DECLARE FLAGS BIT (8) ALIGNED;
```

PL/I allocates storage for an aligned bit-string variable on a byte boundary and reserves an integral number of bytes to contain the variable. Unaligned bit-string variables always occupy only as many bits as are needed to contain them. They need not be on byte boundaries.

In general, operations involving unaligned bit-string variables are less efficient than those involving aligned bit-string variables. Unaligned bit-string variables are also invalid as the targets of the FROM and INTO options of record I/O statements, and as the argument of the ADDR built-in function. Moreover, most non-PL/I programs that accept bit-string arguments require the strings to be aligned.

It is recommended, therefore, that you declare bit-string variables with the ALIGNED attribute in most cases. Use unaligned bit-string variables when bit strings must be packed as tightly as possible, for example, in arrays and in structures.

8.4.4 Bit Strings and Integers

PL/I defines conversions between bit-string data and other data types, and the VAX–11 PL/I compiler carries out these conversions. (Appendix A contains details of the rules governing them.) However, the conversions defined by PL/I are not always straightforward or intuitive. Consider the following example:

```
DECLARE BITSTR BIT (10);
        BITSTR = 1;
        PUT LIST (BITSTR);
```

Its output is

```
'0001000000'B
```

While the result may seem strange, it conforms to PL/I's rules for conversion to bit strings. In this case, the fixed-decimal constant 1 is converted to a FIXED BINARY(4) value, which is in turn converted to an intermediate bit string of length 4:

```
'0001'B
```

Next, this intermediate bit string is assigned to the variable BITSTR. Since BITSTR is of length 10, the intermediate bit string is padded on the right with zeros, producing the result as output by PUT LIST. If you now attempt to interpret the value of BITSTR as an integer (for example, by using BITSTR as the argument of the BINARY built-in function), the result would be 64, not 1.

This example illustrates a general consideration to be kept in mind when using bit-string variables as integers: the padding and truncation that take place during assignment of bit strings of different lengths result in implicit multiplication or division of the bit string's integer value.

One more factor to remember when using bit strings to represent integers is that extra execution time is required to reverse the order of bits when computing the integer's value. Using arithmetic variables to represent integers is more efficient.

The upshot of these considerations is that you should not use bit strings to represent integers (or other data types) unless there is a compelling reason to do so.

8.4.5 Replication Factor for String Constants

A replication factor is an unsigned integer constant that specifies the number of times a simple string constant is replicated. A replication factor permits repetition of character strings and bit strings in any context where a simple string constant is permissible, including format items and assignment, string, and arithmetic operations. The format of a replication factor is

```
(r)'string'
```

r

An unsigned integer that represents the number of times that the string is to be replicated.

string

A simple string constant to be replicated. The string is enclosed in apostrophes.

For example:

```
(4)'season    '
```

This example replicates the string four times. The resulting character constant looks like this:

```
season    season    season    season
```

For example:

```
DECLARE (A) BIT (800);
     A = (400) '2'B2;
     PUT SKIP LIST ((A));
END;
```

In this example, A will be replicated to its maximum specified length of 800 characters.

The resulting character constant looks like this:

```
'10101010101010101010101010101010
     .
     .
     .
1010101010101010101010101010'B
```

A replication factor may be used in combination with the iteration factor in INITIAL. For example, the following two statements are equivalent:

```
INITIAL ((10)('ABCABC'))

INITIAL ((10)((2)'ABC'))
```

The first statement uses an iteration factor exclusively; the second statement combines an iteration factor of 10 with a replication factor of 2. Note than an extra set of parentheses is required to separate the iteration factor from the replication factor and the character string.

Chapter 9

Storage Classes

The storage class to which a variable belongs determines whether PL/I allocates storage for it at compile time or dynamically during the execution of the program, and also indicates the method and the extent of access to the variable. This chapter describes

- Automatic variables—for which PL/I allocates storage upon activation of the declaring procedure.
- Static variables—for which PL/I allocates storage at program activation, and which exist for the duration of the program execution.
- Internal variables—that can be referenced only by the declaring procedure and its dynamic descendants.
- External variables—that can be known to blocks outside the block in which they are declared.
- Based variables—that are allocated dynamically under program control during execution, and that are accessed by means of a pointer.
- Controlled variables—that are allocated dynamically under program control during execution, and that are accessed sequentially, as on a stack.
- Defined variables—for which storage is not allocated, but which share the storage of a specified base variable.

In addition to their storage class, variables can be categorized as addressable and nonaddressable. In some contexts, such as in argument lists of certain built-in functions, a variable must be addressable. A variable is addressable if it has the following properties:

1. It is not suitable for bit-string overlay defining; that is, it does not consist entirely of unaligned bit data.
2. It is not an unconnected array (typically a member of an array of structures).
3. It is not declared with the VALUE attribute.

These rules ensure that the variable can occupy contiguous storage beginning on a byte boundary. (Note that constants are never addressable in PL/I.)

9.1 Automatic Variables

The default storage class attribute for PL/I variables is AUTOMATIC. PL/I does not allocate storage for an automatic variable until the block that declares it is activated. When the block is deactivated, the storage is released.

The storage requirements of an automatic variable are evaluated each time the block is activated. Thus, the length of an automatic character-string variable may be specified as follows:

```
DECLARE STRING_LENGTH FIXED;
    .
    .
    .
COPY: BEGIN;
DECLARE TEXT CHARACTER(STRING_LENGTH);
```

When this begin block is activated, the length of TEXT is evaluated. The variable is allocated storage depending on the value of STRING_LENGTH, which must have a valid value.

9.2 Static Variables

A static variable is allocated storage when the program is activated, and it exists for the duration of the program. A variable has the static attribute if it is declared with any of the attributes STATIC, EXTERNAL, GLOBALDEF, or GLOBALREF. Static arrays and strings must be declared with constant extents.

If a block that declares a static variable is entered more than once during the execution of the program, the value of the static variable remains valid. For example:

```
UNIQUE_ID: PROCEDURE RETURNS (FIXED BINARY(31));
DECLARE ID STATIC INTERNAL FIXED INITIAL (0);
        ID = ID + 1; /* Increment ID */
        RETURN (ID);
        END;
```

The function UNIQUE_ID declares the variable ID with the STATIC attribute and specifies an initial value of zero for it. The variable is initialized to this value when the program is activated. The storage for the variable is preserved, and the function returns a different integer value each time it is referenced.

A variable with the STATIC attribute can also have external scope: that is, its definition and value can be accessed by any other procedure that declares it with the STATIC and EXTERNAL attributes.

9.3 Internal Variables

An internal variable is known only within the block in which it is defined and within all contained blocks. By default, PL/I gives all variables the internal attribute.

9.4 External Variables

An external variable provides a way for external procedures to share common data. All declarations that refer to an external variable must also declare it with the attribute EXTERNAL (or with an attribute that implies EXTERNAL) and with identical data type attributes. Figure 9-1 illustrates how procedures can use external variables.

The format of the EXTERNAL attribute is

```
{ EXTERNAL }
{ EXT      }
```

The EXTERNAL attribute is implied by the FILE, GLOBALDEF, and GLOBALREF attributes, and also by declarations of entry constants (that is, declarations that contain the ENTRY attribute but not the VARIABLE attribute). For variables, the EXTERNAL attribute implies the STATIC attribute.

The following rules apply to the use of external names:

- The EXTERNAL attribute directly conflicts with the AUTOMATIC, BASED, CONTROLLED, and DEFINED attributes.
- The EXTERNAL attribute cannot be applied to minor structures, members of structures, parameters, or descriptions in an ENTRY or RETURNS attribute.
- The EXTERNAL attribute is invalid for variables that are the parameters of a procedure.
- If a variable is declared as EXTERNAL STATIC INITIAL, all blocks that declare the variable must initialize the variable with the same value.

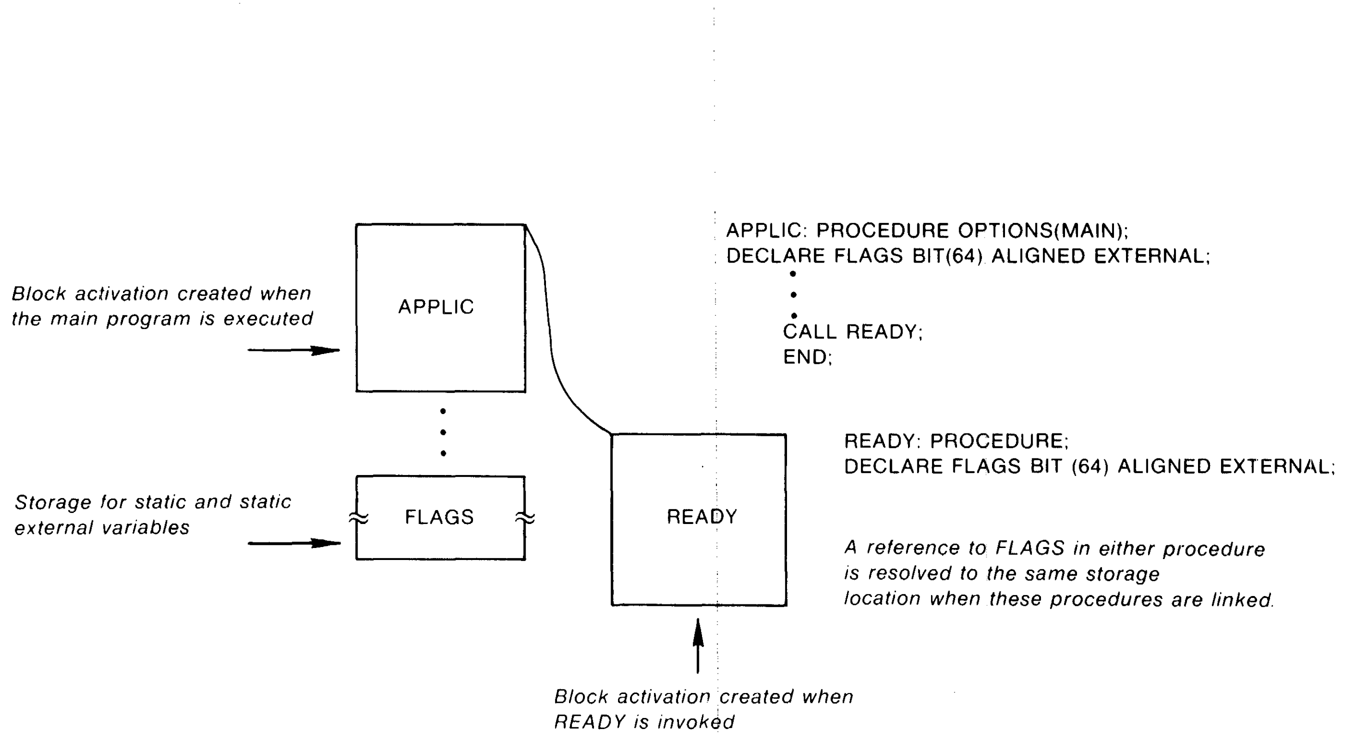


Figure 9-1: External Variables

9.5 Based Variables

The BASED attribute defines a variable whose storage is accessed by means of a pointer. When you declare a based variable, you provide PL/I with a description of the data to be accessed by the variable. The actual data must be referenced by a pointer that contains the address of its storage location. For example:

```
DECLARE BUFFER CHARACTER(80) BASED (BUF_PTR),
        LINE CHARACTER(80),
        BUF_PTR POINTER;

BUF_PTR = ADDR(LINE);
```

The declaration of the variable BUFFER does not allocate any storage for it. Rather, PL/I associates the declaration of the variable with the pointer variable BUF_PTR. During the execution of the program, the value of the pointer variable is set to the location (address) in storage of the variable LINE by means of the ADDR built-in function. This effectively associates the description of BUFFER with the actual data value of LINE.

A based variable can be associated with a storage location by using the ADDR built-in function, as in the preceding example; by the ALLOCATE statement; by a locator-qualified reference to the based variable; by the SET option of the READ statement; or by explicit allocation within an area.

The next sections cover the following topics:

- Data types used with based variables: pointers, areas, and offsets
- Declarations of based variables
- The ALLOCATE and FREE statements, used to obtain and release storage for based variables
- Mechanisms for referring to based variables and for obtaining pointer values to them
- An example of based variables in use
- Rules for data type matching in references to based variables

9.5.1 Data Types Used with Based Variables

The data types most commonly associated with based variables are pointers, areas, and offsets.

A pointer is a variable whose value represents the location in memory of another variable or data item. Pointers are used to access based variables and buffers allocated by the system as a result of the SET option of the READ and ALLOCATE statements.

Areas are regions of storage in which based variables may be allocated and freed. The use of areas can simplify and speed operations involving large or numerous based variables. An offset is a value indicating the location of a based variable relative to the beginning of an area.

9.5.1.1 Pointer Data

Use the POINTER attribute to indicate that the associated variable will identify locations of data. The format of the POINTER attribute is

```
{ POINTER }
{ PTR     }
```

The POINTER attribute conflicts with all other data type attributes.

Expressions containing pointer variables are restricted to the relational operators = and ^=, for testing the equality or inequality of two values. You can also use pointer variables in simple assignment statements that assign a pointer value to a pointer variable. Finally, you can use a pointer variable as the source or target in an assignment statement involving an offset variable or offset value. Section 9.5.1.2 describes these assignments.

9.5.1.2 Area and Offset Data

Areas provide the following programming capabilities:

- You can allocate based variables within a specific area, and assign or transmit the entire area in a single operation. You can refer to the variables by offset values within the area; the offset values remain valid through assignment or transmission.
- You can control the allocation of storage for related variables by placing them in the same area, thus improving the locality of reference. Also, the storage for all allocations within an area may be recovered in one operation by freeing the area itself.
- You can use a structure containing an area to represent a disk file that is mapped into a process's virtual memory space.

It is the responsibility of the user, in the program that declares and allocates an area, to control the allocation of variables within it. The first longword is reserved for future use by DIGITAL; it must be set to zero when the area is initialized, and, remain unmodified.

You define an area by the declaration of a variable with the AREA attribute. An area variable can belong to any storage class. The format of the AREA attribute is

```
AREA (extent);
```

extent

The size of the area in bytes. The extent must be a nonnegative integer value. The maximum size is 500 million bytes. The rules for specifying the extent are as follows:

- For a static variable declaration, extent must be a decimal integer constant.
- In the declaration of a parameter or in a parameter descriptor, extent may be specified as an integer constant or an asterisk (*).
- For an automatic or based variable, extent may be specified as an integer constant or an expression. In the case of automatic variables, the extent expression must not contain any variables or functions declared in the same block, except for parameters.

You can specify an area variable as the target of an assignment statement only in the following case:

area-variable-1 = area-variable-2;

where both areas have the same extent. The complete contents of the source are copied to the target. All other specifications of an area variable as the target of an assignment statement are invalid. An area variable cannot be used in an expression containing operators.

An area can be the source or target of data transmission in either of the record I/O statements READ or WRITE. The complete contents of the area are transmitted.

You declare an offset variable with the OFFSET attribute. The format of the OFFSET attribute is

OFFSET [(area-reference)];

area-reference

The name of a variable with the AREA attribute. The value of the offset variable will be interpreted as an offset within the specified area. You must omit the area reference if the OFFSET attribute is specified within a returns descriptor, parameter declaration, or a parameter descriptor.

For example:

```
DECLARE MAP_SPACE AREA (40960),
        MAP_START OFFSET (MAP_SPACE),
        MAP_LIST(100) CHARACTER(80) BASED (MAP_START);
```

These declarations define an area named MAP_SPACE; an offset variable, MAP_START, that will contain offset values within that area; and a based variable whose storage is located by the value of MAP_START.

Offset variables are given values by assignment from existing offset values or from conversion of pointer values. The OFFSET built-in function (described in Section 19.2) converts a pointer value to an offset value. PL/I also automatically converts a pointer value to an offset value, and vice versa, in an assignment statement. The following assignments are valid:

1. pointer-variable = pointer-value;
2. offset-variable = offset-value;
3. pointer-variable = offset-variable;
4. offset-variable = pointer-value;

In (2), any area references are ignored in the assignment; therefore, the offset value and variable can refer to different areas. In (3) and (4), the offset variable must have been declared with an area reference.

Expressions containing offset variables are restricted to the relational operators = and ^=, for testing the equality or inequality of two values.

9.5.2 Declaring Based Variables

You declare based variables with the BASED attribute. The format is

```
DECLARE variable-reference BASED [ (loc-reference) ];
```

variable-reference

The variable that is to have the BASED attribute. It can be any scalar, array, area, or major structure variable possessing any of the attributes that do not conflict with BASED (listed below).

loc-reference

A reference to a pointer or offset variable or pointer-valued function. If the reference is to an offset variable, that variable must be declared with a base area. Each time a reference is evaluated, that is, to the based variable without an explicit pointer or offset qualifier, loc-reference is evaluated to obtain the pointer or offset value.

The following restrictions apply to declarations of based variables:

- These attributes conflict with the BASED attribute:

AUTOMATIC	INITIAL
CONTROLLED	parameter
DEFINED	READONLY
EXTERNAL	STATIC
GLOBALDEF	VALUE
GLOBALREF	

- The BASED attribute cannot be applied to minor structures or members of structures, parameters, or descriptors in an ENTRY or RETURNS attribute.

9.5.3 ALLOCATE Statement

The ALLOCATE statement obtains storage for a based or controlled variable and optionally sets a pointer variable equal to the address of the storage. Its format is

```
{ ALLOCATE }
{ ALLOC    }  variable-reference [SET(pointer-reference)];
```

variable-reference

A based or controlled variable for which storage is to be allocated. It can be any scalar, array, area, or major structure variable; it must be declared with the BASED or CONTROLLED attribute.

SET(pointer-reference)

The specification of the pointer variable that is assigned the value of the location of the allocated storage. If the SET option is omitted, the based variable must have been declared with BASED(pointer-reference), and the variable designated by that pointer reference is assigned the location of the allocated storage.

A pointer reference cannot be used with controlled variables.

The following example illustrates the declaration and allocation of based variables.

```
DECLARE STATE CHARACTER(100) BASED (STATE_POINTER),
        STATE_POINTER POINTER;

ALLOCATE STATE;
```

This ALLOCATE statement allocates storage for the variable STATE; the pointer STATE_POINTER points to the location of the allocated storage.

An ALLOCATE statement obtains as much storage as is necessary to accommodate the current extent of the specified variable. If, for example, a character-string variable is declared with an expression for its length, the ALLOCATE statement evaluates the current value of the expression to determine the amount of storage to be allocated. For example:

```
DECLARE BUFFER CHARACTER (BUFLEN) BASED,
        BUF_PTR POINTER;
    .
    .
    .
BUFLEN = 80;
ALLOCATE BUFFER SET (BUF_PTR);
```

Here, the value of BUFLEN is evaluated when the ALLOCATE statement is executed. The ALLOCATE statement allocates 80 bytes of storage for the variable BUFFER and sets the pointer variable BUF_PTR to BUFFER's location.

9.5.4 FREE Statement

The FREE statement releases the storage that was allocated for a based variable. Its format is

FREE variable-reference ;

variable-reference

A reference to the based variable whose storage is to be released.

If you do not explicitly free the storage acquired for a based variable, it is not freed until the program terminates.

If you free a variable that is explicitly associated with a pointer, the pointer variable becomes invalid and must not be used to reference storage.

The following examples illustrate the use of the FREE statement.

```
FREE LIST;
FREE P->INREC;
```

These statements release the storage acquired for the based variable LIST and for the allocation of INREC pointed to by the pointer P.

```
ALLOCATE STATE SET (STATE_PTR);
    .
    .
    .
FREE STATE;
```

This FREE statement releases the storage for the based variable STATE and makes the value of STATE_PTR undefined.

9.5.5 Referring to Based Variables

A reference to a based variable (except in an ALLOCATE statement) must specify a pointer or offset reference designating the storage to be accessed. This qualifying pointer or offset reference may be implicit, by specifying it in the BASED attribute, or explicit, by prefixing the based variable reference with a locator qualifier. A complete based variable reference (with the locator qualifier) has the form

qualifying-reference -> base-reference

Whether explicit or implicit, the qualifying reference must be to a pointer variable, a pointer-valued function, or an offset variable declared with a base area. The qualifying reference is evaluated each time the complete

reference is evaluated and must yield a valid pointer value. If the qualifying reference is to an offset variable, the offset value is converted to a pointer using the base area specified in the offset variable's declaration.

You may use both implicit and explicit qualification with the same based variable; the explicit qualifier overrides the implicit one. For example:

```
DECLARE X FIXED BIN BASED(P),
     P POINTER,
     (A,B) FIXED BIN;
P = ADDR(A);
X = ADDR(B)->X;
```

In the second assignment statement, the reference to X on the left-hand side of the assignment has the implicit qualifier P, which is the address of the variable A. The reference to X on the right-hand side is explicitly qualified with the address of another variable, B. This assigns the value of B to the variable A.

In VAX-11 PL/I, a valid pointer value may be obtained in any of the following ways:

- Through the SET option of the ALLOCATE statement
- From a user-provided storage allocation routine
- Through the SET option of the READ statement
- From applying the ADDR built-in function to an addressable variable
- By converting an offset value to a pointer value

A pointer value is valid only as long as the storage to which it applies remains allocated. Moreover, a pointer obtained by applying ADDR to a parameter is valid only as long as the parameter's procedure invocation exists, even though the storage pointed to may exist longer.

The NULL built-in function returns a null pointer value that can be assigned to pointer and offset variables, but that is not valid for qualifying a based variable reference.

When you use the READ statement with a based variable, you do not have to define storage areas within your program to buffer records for I/O operations. If you specify the SET option on the READ statement, the READ

statement places an input record in a system buffer and sets a pointer variable to the location of this buffer. For example:

```
DECLARE REC_PTR POINTER ,
        NEW_BALANCE FIXED DECIMAL (6,2),
        INFILE FILE RECORD INPUT SEQUENTIAL ;
DECLARE 1 RECORD_LAYOUT BASED (REC_PTR),
          2 NAME CHARACTER (15),
          2 AMOUNT PICTURE '999V99',
          2 BALANCE FIXED DECIMAL (6,2);
    .
    .
    .
READ FILE (INFILE) SET (REC_PTR) ;
    .
    .
    .
REC_PTR->BALANCE = NEW_BALANCE;
REWRITE FILE (INFILE);
```

In this example, the structure defined to describe the records in a file is declared with the BASED attribute; the declaration does not reserve storage for this structure. When the READ statement is executed, the record is actually read into a system buffer, and the pointer REC_PTR is set to its location.

When you use the SET option with the READ statement, a subsequent REWRITE statement need not specify the record to be rewritten. PL/I rewrites the record indicated by the pointer variable specified in the READ statement.

The ADDR built-in function returns the storage location of a variable. You can use it to associate the storage occupied by a variable with the description of a based variable. For example:

```
DECLARE A FIXED BINARY BASED (X),
        B FIXED BINARY,
        X POINTER;

X = ADDR (B);
A = 15;
```

In this example, the variable A is declared as a based variable, with X designated as its pointer. The variable B is an automatic variable; PL/I allocates storage for B when the block is activated. When the ADDR built-in function is referenced, it returns the location in storage of the variable B, and the assignment statement gives this value to the pointer X. This assignment associates the variable A with the storage occupied by B. Because A is based on X and X points to B, an assignment statement that gives a value to A actually modifies the storage occupied by the variable B. Figure 9–2 illustrates this example.

DECLARE A CHARACTER (1000) BASED(X);

No storage is allocated for A.

DECLARE B CHARACTER (1000);

B

B is allocated a thousand bytes of storage.

DECLARE X POINTER;

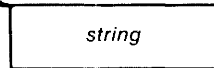
X

X is allocated a longword of storage.

X = ADDR (B);
A = 'STRING';

X

B



The value of X is B's memory location. A reference to A is resolved as a reference to B.

ZK-026-81

Figure 9–2: Using the ADDR Built-In Function

9.5.6 Example of Based Variable Use

The program DEFINED uses based variables and the READ SET statement to process a file of personnel data (PERSONNEL.DAT). The file has two types of valid records, a pay record and a health record, which are identified by a 1-character code in the first position. The two record types are declared as based structures (PAY_RECORD and HEALTH_RECORD), one of which is selected based on the record type character ('P' for pay, 'E' for health). Any record that does not begin with one of these characters is invalid and is written out as a reference to the based character variable INVALID_RECORD.

```
DEFINED: PROCEDURE OPTIONS(MAIN);

DECLARE P POINTER; /* pointer to structures */

DECLARE 1 PAY_RECORD BASED(P),
          2 RECORD_TYPE CHARACTER(1),
          2 NAME CHARACTER(20),
          /* the two structures differ in this member: */
          2 GROSS_PAY PICTURE '999999V.99';

DECLARE 1 HEALTH_RECORD BASED(P),
          2 RECORD_TYPE CHARACTER(1),
          2 NAME CHARACTER(20),
          2 EXAM_DATE CHARACTER(9);

DECLARE INVALID_RECORD CHARACTER(30) BASED(P);

DECLARE PERSONNEL RECORD FILE;
DECLARE PERSOUT STREAM OUTPUT PRINT FILE;

/* used to control DO group: */
%REPLACE NOTENDFILE BY '1'B;

ON ENDFILE(PERSONNEL) BEGIN;
   PUT FILE(PERSOUT) SKIP LIST
           ('All processing complete.');
   STOP;    /* program stops here */
   END;

OPEN FILE(PERSONNEL) INPUT TITLE('PERSONNEL.DAT');

DO WHILE(NOTENDFILE);
/* terminated by ENDFILE ON-unit */

READ FILE(PERSONNEL) SET(P);
/* P is the location of the
record acquired by the READ statement */

IF P->PAY_RECORD.RECORD_TYPE = 'P' THEN
   PUT FILE(PERSOUT) SKIP LIST
          ('Name=',P->PAY_RECORD.NAME,
          'Gross pay=',P->GROSS_PAY);
```

```
ELSE        /* either a health record or an invalid record */
   DO;
   IF P->HEALTH_RECORD.RECORD_TYPE = 'E' THEN
   PUT FILE(PERSOUT) SKIP LIST
          ('Name=',P->HEALTH_RECORD.NAME,
          'Exam date:',P->EXAM_DATE);
   ELSE /* invalid record type */
   PUT FILE(PERSOUT) SKIP LIST
            ('Invalid record:',P->INVALID_RECORD);
   END;

END; /* repeat DO group until ENDFILE is signaled */

END DEFINED;
```

For example, if the file PERSONNEL.DAT contains these records:

```
PMary A. Ford         125000.55
EMary A. Ford         22July 80
t12345678901234567890pppppp.pp
```

then the output file (PERSOUT.DAT) will contain the following output:

```
Name=   Mary A. Ford                Gross pay=      125000.55
Name=   Mary A. Ford                Exam date:      22July 80
Invalid record: t12345678901234567890pppppp.pp
All processing complete.
```

Notice these other features of the program:

- The references to based variables have a locator qualifier (P->) for clarity. However, since all were declared with P as their pointer reference, the locator qualifier could have been omitted.
- References to the structure members RECORD_TYPE and NAME must be fully qualified with the name of their containing structures (PAY_RECORD and HEALTH_RECORD) because both structures have members with these names. In contrast, GROSS_PAY and EXAM_DATE are unique to their structures and need not be fully qualified.

9.5.7 Data Type Matching for Based Variables

In most applications, the data type of a based variable reference is identical to the data type under which the accessed storage is allocated. (The *VAX-11 PL/I Encyclopedic Reference* contains a precise definition of identical data types.) However, it is not required that the data types be identical. In standard PL/I, it is sufficient that the data types match as for overlay defining or that they are left-to-right equivalent. In VAX-11 PL/I, the data types may be quite different, although the program will then depend on the VAX-11 internal representation of data. The following sections discuss type-matching criteria in more detail.

9.5.7.1 Matching by Overlay Defining

This type of matching is in effect if the based variable reference and the variable for which the storage was originally allocated are both suitable for character- (or bit-) string overlay defining. (See Section 9.7 for a discussion of string overlay defining.) The only further restriction is that the size n (in characters or bits) of the based variable reference must be less than or equal to the size in characters or bits of the original variable. The based variable reference accesses the first n characters or bits of the storage.

The program in the preceding section contains an example of this type of matching. The structure members PAY_RECORD.GROSS_PAY (a character string) and HEALTH_RECORD.EXAM DATE (a picture) are not identical data types. However, both are stored as a character string of length 9; therefore, they meet the criteria for string overlay defining and for data type matching.

9.5.7.2 Matching by Left-to-Right Equivalence

This type of matching applies to structured variables that are identical up to a certain point. To see if this applies, examine the declaration of the based variable, and consider only the portion on the left that includes the referenced member and all of the level-2 substructure containing the referenced member (if the member is not itself at level 2). If the original variable's declaration has a similar left part with identical data type, then the based variable reference and the original reference match. For example:

```
DECLARE 1 S1 BASED (P),
          2 X,
            3 (A,B) FIXED BIN,
          2 Y,
            3 C CHAR(10),
            3 D(5) FLOAT;

DECLARE 1 S2 BASED(P),
          2 X,
            3 (A,B) FIXED BIN,
          2 Y,
            3 C CHAR(10),
            3 E BIT(32);

ALLOCATE S1;

S2.A = 3; /* valid l-to-r match */
S2.C = 'X'; /* INVALID */
```

In the first assignment, S2.A is a valid reference because S1 and S2 match through the level-2 structure X. In the second assignment, S2.C is invalid in standard PL/I because the level-2 structures S2.Y and S1.Y do not match. (However, the reference to S2.C does work in VAX-11 PL/I.)

This sort of matching is useful in connection with data structures and files, where the first part of a record contains a value indicating the precise structure of the remainder of the record.

9.5.7.3 Nonmatching Based Variable References

In VAX-11 PL/I, a based variable reference need not match the variable for which the storage was originally allocated. The only requirement is that the size of the based variable in bits be less than or equal to the size of the original variable in bits. However, use of such nonmatching references requires knowledge of the VAX-11 internal representation of data, and you should not expect the resulting code to be transportable to other PL/I implementations. For example:

```
DECLARE X FLOAT BINARY(24);
DECLARE 1 S BASED(ADDR(X)),
          2 FRAC_1 BIT(7),
          2 EXP BIT(8),
          2 SIGN BIT(1),
          2 FRAC_2 BIT(16);

EXP = '0'B; /* set exponent to 0 */
SIGN = '1'B; /* set sign negative */
X = X + 1;
```

The declaration of S describes the internal representation of a VAX-11 single-precision floating-point number. The first two assignments set the sign and exponent fields to the reserved operand combination; the assignment to X causes a reserved operand exception.

9.6 Controlled Variables

A controlled variable is a variable whose actual storage is allocated and freed dynamically in "generations," of which only the most recent is accessible to the program. Controlled variables must be defined by the CONTROLLED attribute, which has the format

DECLARE variable-reference { CONTROLLED | CTL } ;

variable-reference

The variable that is to have the CONTROLLED attribute. It can be any scalar, array, area, or major structure variable possessing any of the attributes that do not conflict with CONTROLLED (listed below).

The following restrictions apply to declarations of controlled variables:

- These attributes conflict with the CONTROLLED attribute:

  AUTOMATIC INITIAL
  BASED parameter
  DEFINED READONLY
  GLOBALDEF STATIC
  GLOBALREF VALUE

- The CONTROLLED attribute cannot be applied to minor structures, members of structures, parameters, or descriptions in an ENTRY or RETURNS attribute.

A controlled variable has no storage assigned to it until an ALLOCATE statement allocates storage for it. Each storage assignment is a "generation" of the variable. Subsequent ALLOCATE statements allocate subsequent generations. At any time in the program's execution, a reference to a controlled variable is a reference to the most recent generation of that variable, that is, the generation created by the most recent ALLOCATE statement.

The FREE statement frees the most recent generation of a controlled variable. If an attempt is made to free a controlled variable for which no generation exists (or to refer to such a variable), PL/I signals the ERROR condition.

The following example illustrates the use of controlled variables:

```
CONT: PROCEDURE OPTIONS (MAIN);

DECLARE STR CHARACTER (10) CONTROLLED;

    ALLOCATE STR;
    STR = 'First';
    ALLOCATE STR;
    STR = 'Second';
    ALLOCATE STR;
    STR = 'Third'
    PUT SKIP LIST (STR);
    FREE STR;
    PUT SKIP LIST (STR);
    FREE STR;
    PUT SKIP LIST (STR);
    FREE STR;

END;
```

The output of this program is

```
Third
Second
First
```

9.6.1 Using the ALLOCATION Built-In Function

Since only the most recent generation of a controlled variable is available to a program, controlled variables provide an easy way to implement a stack. The ALLOCATE statement is equivalent to a push operation and the FREE statement is equivalent to a pop operation. The ALLOCATION built-in function returns the number of generations of a variable, so it can be used to find out if the stack is empty. For example:

```
DECLARE NEXT_MOVE CHARACTER(5) CONTROLLED,
    DIRECTIONS(4) CHARACTER(5) INITIAL(
    'North','East','South','West'),
    D FIXED BINARY (7);
        .
        .
        .
    ALLOCATE NEXT_MOVE;          /* Part of a loop that reports */
    NEXT_MOVE = DIRECTIONS(D);   /* moves in reverse order */
        .
        .
        .
    DO WHILE                     /* Print moves in correct order */
        (ALLOCATION(NEXT_MOVE) ^= 0);
        PUT SKIP LIST ('Go ', NEXT_MOVE);
        FREE NEXT_MOVE;
    END;
```

9.6.2 Using the ADDR Built-In Function

A controlled variable can be used as the argument of the ADDR built-in function. If a generation exists, ADDR returns a pointer to it. If no generation of the variable exists, ADDR returns the null pointer. Thus, ADDR can be used to preserve a pointer to a generation of a controlled variable that later becomes "hidden" under further generations, as in the following example:

```
DECLARE STOPS CHARACTER (20) VARYING CONTROLLED,
    MIDPOINT CHARACTER (20) VARYING BASED (P),
    P POINTER;
        .
        .
        .
    ALLOCATE STOPS;
    STOPS = CURRENT_LOC;
    IF I = 5 THEN P = ADDR(STOPS);
        .
        .
        .
    PUT SKIP LIST (
        'End reached!  Halfway point was', MIDPOINT);
```

At a certain point during the execution of this program, the ADDR built-in function captures the address of the current generation of STOPS and assigns it to P. Later, after more generations of STOPS have been allocated, MIDPOINT (which is based on P) has the value of that same intermediate generation of STOPS.

Note, however, that the value of P (and therefore of MIDPOINT) is valid only so long as the intermediate generation of STOPS to which P points is allocated. As soon as that generation is freed, the value of P becomes invalid, and it must not be used in a pointer-qualified reference until it is reassigned.

A controlled variable cannot be used in a pointer-qualified reference. In the example above, a reference like the following would be illegal:

```
P->STOPS
```

9.7 Defined Variables

The DEFINED attribute indicates that PL/I is not to allocate storage for the variable, but is to map the description of the variable onto the storage of a base variable. The DEFINED attribute provides a way to access the same data using different names. Its format is

```
{ DEFINED }
{ DEF     }  (variable-reference) [ POSITION (expression) ]
```

variable-reference

A reference to a base variable that has storage associated with it. It must not have the BASED or DEFINED attribute. The base variable and the declared variable must satisfy the rules given later in this section.

POSITION (expression)

Specifies the character or bit position in the base variable at which the defined variable begins. The expression is an integer expression that specifies a position in the base. A value of 1 indicates the first character or bit. You can only use the POSITION attribute when the defined variable satisfies the rules for string overlay defining (described later in this section).

The following restrictions apply to defined variables:

- These attributes conflict with the DEFINED attribute

AUTOMATIC	GLOBALDEF
BASED	GLOBALREF
CONTROLLED	READONLY
EXTERNAL	STATIC
INITIAL	parameter
VALUE	

- The DEFINED attribute cannot be applied to minor structures, members of structures, parameters, or descriptions in an ENTRY or RETURNS attribute.

When you use the DEFINED attribute in the declaration of a variable, PL/I associates the description of the variable in the declaration with the storage allocated for the variable on which the declaration is defined. For example:

```
DECLARE NAMES(10) CHARACTER(5) DEFINED (LIST),
        LIST(10) CHARACTER(5);
```

In this example, the variable NAMES is a defined variable; its data description is mapped to the storage occupied by the variable LIST. Any reference to NAMES or to LIST is resolved to the same location in memory.

With defined variables that meet the criteria for string overlay defining (described below), you can use the POSITION attribute to specify the position in the base variable at which the definition begins. For example:

```
DECLARE ZIP CHARACTER(20),
        ZONE CHARACTER(10)
        DEFINED(ZIP) POSITION(4);
```

This statement declares the variable ZONE and maps it to characters 4 through 13 of the variable ZIP.

The extent of a defined variable is determined at the time of block activation, but the base reference (and the position, if the POSITION attribute is also specified) is interpreted each time the defined variable is referenced. For example:

```
DECLARE I FIXED,
    A(10) FIXED,
    B FIXED DEFINED(A(I));
DO I = 1 TO 10;
B = I;
END;
```

The DO group assigns I to A(I) for I = 1,2,...10.

The base reference of a defined variable may not be a reference to a based variable or to another defined variable. A defined variable and its base reference must satisfy one of the following criteria:

- They must have identical data types (see the *VAX-11 PL/I Encyclopedic Reference* for a precise definition).
- They must both be suitable for character-string overlay defining.
- They must both be suitable for bit-string overlay defining.

If the defined variable is specified with the POSITION attribute, then both the defined variable and the base reference must be suitable for bit- or character-string overlay defining.

In brief, a variable is suitable for overlay defining if it consists entirely of characters or bits, and those characters or bits are packed into adjacent storage without gaps. (The *VAX-11 PL/I Encyclopedic Reference* contains precise rules.) Such a variable can be treated as a string or interpreted as different types of aggregates. For example, Figure 9-3 shows a 50-byte region of storage treated either as a 10-element array (A) of 5-character strings or as a 5-element array (B) of 10-character strings.

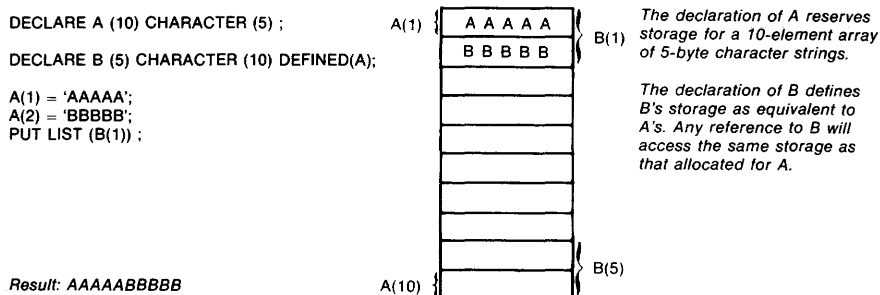


Figure 9-3: An Overlay Defined Variable

Chapter 10

Aggregates

Aggregates are groupings of variables. There are two types of aggregate:

- An array is an aggregate in which all items, called elements, have the same data type. An individual element of an array is referred to by an integer subscript that designates the element's position, or order, in the array. Elements can be scalar data items or structures.
- A structure is an aggregate in which individual items, called members, can have different data types. Individual members are referred to by qualified references that give, in general, the names of the structure itself and of the individual member.

Aggregates can also be formed from arrays whose elements are structures, or from structures whose individual members are arrays.

10.1 Arrays

Arrays provide an orderly way to manipulate related variables of the same data type. An array variable is defined in terms of the number of elements that the array contains and the organization of those elements. These attributes of an array are called its dimensions.

10.1.1 Array Declarations

The declaration of an array specifies its dimensions, the bounds of each dimension, and the attributes of the elements. The format of a DECLARE statement for an array is

```
DECLARE identifier (bound-pair,...) [attribute ...];
```

or

```
DECLARE (declaration,...) (bound-pair,...) [attribute ...];
```

where each bound pair has the format

```
[lower-bound:]upper-bound
```

or

```
*
```

One bound pair is specified for each dimension of the array, to define the number of elements in that dimension. The extent of an array is the product of the numbers of elements in its dimensions. If omitted, the lower bound is 1 by default.

You can use the asterisk (*) as the bound pair when you declare arrays as parameters of a procedure; the asterisk indicates that the parameter can accept array arguments with any number of elements. (If one dimension is specified with an asterisk, all must be specified with asterisks.)

For example, the statement

```
DECLARE SALARIES(100) FIXED DECIMAL(7,2);
```

declares a 100-element array with the identifier SALARIES. Each element is a fixed-point decimal number with a total of seven digits, two of which are fractional. The statement

```
DECLARE GAME-BOARD(8,8) FIXED BINARY(7);
```

declares a two-dimensional array of 64 integers. The statement

```
DECLARE PM_HOURS(13:24) CHARACTER(2);
```

declares a one-dimensional array of 12 character strings. The elements of the array are numbered 13 through 24 instead of 1 through 12.

You can replace the identifier in a statement with a list of declarations, thereby declaring several arrays with the same attributes. For instance

```
DECLARE (SALARIES,PAYMENTS)(100) FIXED DECIMAL(7,2);
```

declares SALARIES and another array, PAYMENTS, with the same dimensions and other attributes.

The following rules apply to specifying the dimensions of an array and the bounds of a dimension:

- The maximum number of dimensions that an array can have is eight.
- The values you can specify for bounds are restricted as follows:
  - If the array has the STATIC attribute, you must specify all bounds as restricted integer expressions. A restricted integer expression yields only integral results and has only integral operands. Only the operators +, -, *, and the DIVIDE built-in function can be used in such an expression.
  - If the array has the AUTOMATIC, BASED, CONTROLLED, or DEFINED attribute, you can specify the bounds as optionally signed integer constants or as expressions that yield integer values

at run time. If the array has AUTOMATIC or DEFINED, the expressions must not contain any variables or functions that are declared in the same block, except for parameters.

  - If an array is a parameter, you can specify the bounds using optionally signed integer constants or asterisks (*). If you specify any bound as an asterisk, you must specify all bounds with asterisks. An array parameter declared this way inherits the dimensions of the corresponding argument.

- The value of the lower bound you specify must be less than the value of the upper bound.

10.1.2 References to Individual Elements

You refer to an individual element in the array by means of subscripts. Since an array's attributes are common to all of its elements, a subscripted reference has the same properties as a reference to a scalar variable with those attributes.

You must enclose subscripts in parentheses in a reference to an array element. For example, in a one-dimensional array named ARRAY declared with the bounds (1:10), the elements are numbered 1 through 10 and are referred to as ARRAY(1), ARRAY(2), ARRAY(3), and so on. The lower and upper bounds that you declare for a dimension determine the range of subscripts you can specify for that dimension.

For multidimensional arrays, the subscript values represent an element's position with respect to each dimension in the array. In subscripted references for multidimensional arrays, the number of subscripts must match the number of dimensions of the array, and must be separated by commas.

You can specify the subscript of an array element using any variables or expressions having integer values, that is, values that can be expressed as fixed binary or fixed decimal with a zero scale factor. For example:

```
DECLARE DAYS_IN_MONTH(12) FIXED BINARY;

DECLARE (COUNT, TOTAL) FIXED BINARY;
TOTAL = 0;
DO COUNT = 1 TO 12;
      TOTAL = TOTAL + DAYS_IN_MONTH(COUNT);
      END;
```

Here, the variable COUNT is used as a control variable in a DO loop. As the value of COUNT is incremented from 1 to 12, the value of the corresponding element of the array DAYS_IN_MONTH is added to the value of the variable TOTAL.

10.1.3 Initializing Arrays

The INITIAL attribute can be specified for arrays. For example:

```
DECLARE MONTHS (12) CHARACTER (9) VARYING
    INITIAL ('January', 'February', 'March', 'April',
        'May', 'June', 'July', 'August',
        'September', 'October', 'November', 'December');
```

Each element of the array MONTHS is assigned a value according to the order of the character-string constants in the initial list: that is, MONTH(1) is assigned the value 'January'; MONTH(2) is assigned the value 'February'; and so on.

If the array being initialized is multidimensional, the initial values are assigned in row-major order (see Section 10.1.5).

When you want to assign identical initial values to some or all elements of an array, you can use an iteration factor with the INITIAL attribute. For example:

```
DECLARE TEST_AVGS (30,4) FIXED DECIMAL (5,2)
    STATIC INITIAL ((120) 50);
```

This statement declares the array TEST_AVGS with 120 elements, each of which is given an initial value of 50.

Although VAX-11 PL/I supports the initialization of automatic arrays with the INITIAL attribute, it is not always the most efficient way (in terms of program compilation and execution) to initialize array elements. The following notes cover the things you should consider:

- When you initialize elements in an array that has the AUTOMATIC attribute, the compiler does not check that all elements are initialized until run time. Thus, you do not receive any compile-time checking of initialization, even if you used constants to specify the array bounds and iteration factors.
- Your programs will run more efficiently if you initialize automatic arrays with assignment statements rather than the INITIAL attribute.
- If the array is not modified by your program, you can increase program efficiency by declaring the array with the STATIC and READONLY attributes and using the INITIAL attribute to initialize its elements. In this case, the compiler checks that you have initialized all the elements and that they are valid.

Section 11.1.5 provides full details about the INITIAL attribute.

10.1.4 Assigning Values to Array Variables

You can specify an array variable as the target of an assignment statement in the following cases:

- array-variable = expression;

  where the expression yields a scalar value. Every element of the array is assigned the resulting value. The array variable must be a connected array whose elements are scalar.

  Note that the arithmetic operators, such as + and -, cannot have arrays as operands. An assignment of the form

  ```
  ARRAYC = ARRAYA + ARRAYB;
  ```

  is invalid.

- array-variable-1 = array-variable-2;

  where the specified array variables have identical data type attributes and dimensions. Each element in array-variable-1 is assigned the value of the corresponding element in array-variable-2. In this type of assignment, both arrays must be connected. The actual storage they occupy must not overlap, unless the arrays are identical.

All other specifications of an array variable as the target of an assignment statement are invalid.

When you specify an array variable name in the input-target list of a GET LIST or GET EDIT statement, elements of the array are assigned values from the data items in the input stream. For example:

```
DECLARE VERBS (6) CHARACTER (15) VARYING;
GET LIST (VERBS);
```

When this GET LIST statement executes, it accepts data from the default input stream. Each blank-, tab-, or comma-delimited input field is considered a separate string. The values of these strings are assigned to elements of the array VERBS in the order VERBS(1), VERBS(2),...VERBS(6). If a multidimensional array appears in an input-target list, input data items are assigned to the array elements in row-major order (see Section 10.1.5).

An array can also appear, with similar effects, in the output-source list of a PUT statement.

10.1.5 Order of Assignment and Output for Multidimensional Arrays

When a multidimensional array is initialized, or when it is assigned values without references to specific elements, PL/I assigns the values in row-major order. In row-major order, the rightmost subscript varies the most rapidly. For example, an array can be declared as follows:

```
DECLARE TESTS (2,2,3);
```

If TESTS is specified in a GET statement or in a declaration with the INITIAL attribute, values are assigned to the elements in the following order:

TESTS (1,1,1)
TESTS (1,1,2)
TESTS (1,1,3)
TESTS (1,2,1)
TESTS (1,2,2)
TESTS (1,2,3)
TESTS (2,1,1)
TESTS (2,1,2)
TESTS (2,1,3)
TESTS (2,2,1)
TESTS (2,2,2)
TESTS (2,2,3)

When an array is output with a PUT statement, PL/I uses the same order to output the array elements. For example:

```
PUT LIST (TESTS);
```

This PUT statement outputs the contents of TESTS in the order shown above.

10.2 Structures

A structure is a data aggregate consisting of one or more members. The members may be scalar data items, arrays of scalar data items, structures, or arrays of structures, and different members may have different data types. Structures are useful when you want to group related data items having different data types.

A structure declaration defines a structure variable by means of level numbers. For example:

```
DECLARE 1 TRANSACTION,
          2 PART_NUMBER,
            3 FACTORY CHARACTER (3),
            3 ITEM CHARACTER (5),
          2 IN_STOCK BIT (1);
```

The level number 1 indicates that TRANSACTION is a structure variable. TRANSACTION is the name of the entire, or "major," structure. The relationship of the higher numbers (2 and 3) indicates that the associated identifiers are the names of members of the structure TRANSACTION or its "minor" structure, PART_NUMBER.

10.2.1 Structure Declarations

The declaration of a structure defines its organization and the names of members at each level in the structure. The format of a DECLARE statement for a structure is

```
{ DECLARE }  declaration[,...]
{ DCL     }
```

where each declaration is

```
level identifier [(bound-pair,...)] [attribute ...]
```

or

```
level (declaration,...) [(bound-pair,...)] [attribute ...]
```

Each declaration specifies a member of the structure and must be preceded by a level number. As shown, a single variable can be declared at a particular level; or the level can contain one or more complete declarations, including declarations of arrays or of other structures. The major structure name is declared as structure level 1; minor members must be declared with level numbers greater than 1. For example, the statement

```
DECLARE 1 PAYROLL,
          2 NAME,
            3 LAST CHARACTER(80) VARYING,
            3 FIRST CHARACTER(80) VARYING,
          2 SALARY FIXED DECIMAL(7,2);
```

declares a structure named PAYROLL. You can access the last name with a qualified reference:

```
PAYROLL.NAME.LAST = 'ROOSEVELT';
```

Alternatively, since the last and first names have the same attributes, you can declare the same structure as

```
DECLARE 1 PAYROLL,
          2 NAME,
            3 (LAST,FIRST) CHARACTER(80) VARYING,
          2 SALARY FIXED DECIMAL(7,2);
```

The following additional rules apply to the specification of level numbers:

- Level numbers must be specified using decimal integer constants.
- A level number must be separated from its associated variable name by at least one space or tab character.

- Level numbers after level 1 can have any integer value, as long as each level number is equal to or greater than the level number of the preceding level. (There can be only one level 1.)
- Each identifier in the structure must be separated from the declaration of the previous identifier by a comma.
- Substructures at the same logical level of nesting do not have to have the same level number.
- The deepest possible logical level is 15.
- The largest possible level number constant is 32767.

Within a structure, only members at the lowest level of each substructure can be declared with data type attributes. Additional rules for specifying attributes for the various components of a structure are as follows:

- Only these attributes are valid for the major structure name:

  AUTOMATIC EXTERNAL
  BASED INTERNAL
  CONTROLLED STATIC
  DEFINED

- The major structure, a minor structure, or any member of the structure can be dimensioned: that is, there can be arrays of structures and structures whose members are arrays.
- Member names cannot have any of these attributes:

  AUTOMATIC GLOBALDEF
  BASED GLOBALREF
  CONTROLLED READONLY
  DEFINED STATIC
  EXTERNAL VALUE

- If a structure has the STATIC attribute, the extents of all members (that is, lengths for character- and bit-string variables, dimensions for array variables, and area extents) must be specified using optionally signed decimal integer constants.

A structure can be initialized by giving the INITIAL attribute to its members. Not all members need be initialized. For example:

```
DECLARE 1 COUNTS,
         2 FIRST FIXED BIN(15) INITIAL(0),
         2 SECOND FIXED BIN(15),
         2 THIRD (5) FIXED BIN(15) INITIAL (5(1));
```

The first and third members of the structure COUNTS are initialized.

The INITIAL attribute cannot be applied, however, to a major or a minor structure name.

10.2.2 Member Attributes

VAX-11 PL/I supports three "member attributes" so named because they apply specifically to the declaration of structure members rather than to the structure as a whole. Member attributes may be applied to major or minor members of a structure. They are

- The LIKE attribute
- The REFER option
- The UNION attribute

Each is discussed in detail in the following sections.

10.2.2.1 Using the LIKE Attribute

The LIKE attribute copies the member declarations contained with a major or minor structure declaration into the structure variable to which it is applied. The format of the LIKE attribute is

```
level-number identifier [attributes] LIKE reference
```

level-number

Gives the level number to which the declarations in the reference are copied.

identifier

Names the variable to which the declarations in the reference are to be copied.

attribute

Storage class or dimensions appropriate for the level number. You may specify a storage class and dimensions with a minor structure.

reference

The name of a major or minor structure that is known in the current block.

You can use the LIKE attribute to copy the declaration of a major or minor structure to another structure variable. The LIKE attribute copies the logical structuring and member declarations from the major or minor structure to the target variable, but does not copy any storage class attributes or dimensioning (except for dimensioning that is applied to members).

The identifier names the variable to which the declarations in the reference are copied. The reference is the name of a major or minor structure known

to this block. Note that the identifier must be preceded by a level number. Any attributes which can be used with a structure variable at that level can be used with the identifier; for example, a major structure can specify a storage class and dimensions, and a minor structure can specify dimensions.

The following example illustrates the LIKE attribute:

```
DECLARE 1 RES_DATA BASED (RPTR),
          2 DATE CHARACTER(8),
          2 HOTEL_CODE CHARACTER(3),
          2 PARTY_NAME,
            3 LAST CHARACTER(20),
            3 FIRST CHARACTER(10),
          2 STAY FIXED BIN(7),
        1 NEW_RESER LIKE RES_DATA,
            .
            .
            .
    GET LIST (NEW_RESER.DATE,NEW_RESER.HOTEL_CODE);
        .
        .
        .
    RES_DATA = NEW_RESER;
```

In this example, the declaration of NEW_RESER uses the LIKE attribute to create a set of member declarations that duplicate those in RES_DATA. The declaration of NEW_RESER is equivalent to the following:

```
DECLARE 1 NEW_RESER,
          2 DATE CHARACTER(8),
          2 HOTEL_CODE CHARACTER(3),
          2 PARTY_NAME,
            3 LAST CHARACTER(20),
            3 FIRST CHARACTER(10),
          2 STAY FIXED BINARY(7);
```

After the various members of NEW_RESER are assigned data and that data is validated, the entire contents of NEW_RESER are assigned to RES_DATA. This is possible because the two structures are identical, which is a byproduct of using the LIKE attribute.

10.2.2.2 Using the REFER Option

The REFER option is provided in PL/I to create self-defining BASED structures. That is, the value of one member of a based structure is used to determine the size of the storage space allocated for another member of the same structure. The REFER option may be used in a DECLARE statement to specify array bounds, the length of a string, or the size of an area. The format of the REFER option is

refer-element REFER (refer-object-reference)

refer-element

An expression that represents the value assigned to the refer object when the structure is allocated. The refer element must satisfy the following:

- It must be an expression which produces a FIXED BINARY(31) value or a value that can be converted to FIXED BINARY (31).
- It may not reference storage in the structure containing the refer element.

refer-object-reference

A reference to a scalar variable. The refer object reference must satisfy the following:

- It cannot be a subscripted variable reference.
- It cannot be locator qualified.
- It must reference a refer object that is a previous member of the structure containing the REFER option.

refer-object

A scalar variable contained by the structure. The refer object must satisfy the following:

- It must be a previous member of the structure containing the REFER option which references the refer object.
- It must be scalar; it cannot be dimensioned or a dimensioned array.
- It must have a computational data type.

A structure declaration containing the REFER option has the basic format

```
DECLARE 1 STRUCTURE S BASED(P),
          2 I FIXED BINARY(31),
          2 A CHARACTER(20 REFER(I)),
```

where I is the refer object, 20 is the refer element, and I is the refer object reference.

In order to allocate storage for a BASED structure, the structure must have a known size. In the above example, the initial length for A is taken from the refer element, 20. However, the REFER option permits the size of the structure to change at run time as the value of the refer object changes. After allocation, the length of A is determined by the refer object, I.

Multiple REFER options are premitted within a structure.

The following example and diagrams illustrate storage mapping using the REFER option.

```
DECLARE 1 S BASED (POINTER),
          2 I FIXED BINARY(15),
          2 J FIXED BINARY(15),
          2 A CHARACTER ((X*2+2) REFER(I)),
          2 B(2) CHARACTER (Y REFER(J));

ALLOCATE S;
X = 5;
Y = 10;

S.A = 'ABCDEFGHIJKL';
S.B(1) = '0123456789';
S.B(2) = 'NOW IS THE';
     .
     .
     .
END;
```

When this structure is allocated, the refer elements ((X*2+2) and Y) are evaluated and used to determine the length of the associated string. The evaluated refer element value (X*2+2) is assigned to the refer object I and Y is assigned to J. Thereafter, the size of strings A and B are determined by the value of the refer objects I and J.

Storage for the above structure would look like this:

S.I	12	
S.J	10	
S.A	B	A
	D	C
	F	E
	H	G
	J	I
	L	K
S.B(1)	1	0
	3	2
	5	4
	7	6
	9	8
S.B(2)	O	N
		W
	S	I
	T	
	E	H

ZK-1303-83

If the refer objects, I and J, were assigned the following values

```
I = 6;
J = 4;
```

the resulting storage would be remapped as this:

S.I	6	
S.J	4	
S.A	B	A
	D	C
	F	E
S.B(1)	H	G
	J	I
	L	K
S.B(2)	1	0

ZK-1304-83

Note that VAX-11 PL/I does not restrict the use of the REFER option within structure declarations: therefore, exercise caution in its use. If you change a value that causes the size of one or more structure members to decrease, then some storage at the end of the allocated storage will become inaccessible for future reference.

If the scalar variable (the refer object) does not satisfy the following, the results are undefined:

- It must not be assigned a value which is less than zero or greater than the refer element value used for structure allocation.
- It must have the value used for allocation, if the structure is freed.

The following additional rules apply to structures containing the REFER option: .

- A structure containing the REFER option may not be the target of a LIKE reference.
- When a BASED structure is allocated, the order in which the refer elements are selected for evaluation is undefined.
- When a BASED structure is allocated, the order in which the refer objects are selected for initialization is undefined.

10.2.2.3 Using the UNION Attribute

A union is a variation of a structure in which all immediate members occupy the same storage. The UNION attribute (which must be associated

with a level number in a structure declaration) declares a union. All immediate members of the union—that is, all members having a level number one higher—occupy the same storage. A reference to one member of a union refers to storage occupied by all members of the union. Therefore, a union provides a convenient way to look at a large entity (such as a character string or a bit mask) as a series of smaller entities (such as component character strings or individual flag bits).

The UNION attribute is not part of the PL/I G subset; it is provided in VAX-11 PL/I to give users convenient access to data as it is internally represented. Potential applications of unions might depend on the internal representation of data, and would therefore not be transportable to other implementations of PL/I, given that these implementations had the UNION attribute.

The format for using the UNION attribute is

level-number identifier [storage class] UNION

level-number

The level number of the variable with which the declarations in the reference share storage.

identifier

The name of the variable with which the declarations in the reference share storage. A variable declared with the UNION attribute must be a major or minor structure. All members of a UNION must have a constant size.

storage class

The storage class specified for the structure.

The following example illustrates unions:

```
DECLARE 1 CUSTOMER_INFO,
            .
            .
            .
          2 PHONE_DATA UNION,
            3 PHONE_NUMBER CHARACTER (13),
            3 COMPONENTS,
              4 LEFT_PAREN CHARACTER (1),
              4 AREA_CODE CHARACTER (3),
              4 RIGHT_PAREN CHARACTER (1),
              4 EXCHANGE CHARACTER (3),
              4 HYPHEN CHARACTER (1),
              4 SPECIFIC_NUMBER CHARACTER (4),
          2 ADDRESS_DATA,
            .
            .
            .
```

The UNION attribute associated with the declaration of PHONE_DATA signifies that PHONE_DATA's immediate members (PHONE_NUMBER and COMPONENTS) occupy the same storage. Any modification of PHONE_NUMBER also modifies one or more members of COMPONENTS; conversely, modification of a member of COMPONENTS also modifies PHONE_NUMBER. Note, however, that the UNION attribute does not apply to the members of COMPONENTS, since they are not immediate members of PHONE_DATA. The members of COMPONENTS occupy separate storage in the normal fashion for structure members.

Unions provide capabilities similar to those provided by defined variables. However, the rules governing defined variables are more restrictive than those governing unions. The following example demonstrates a use of a union that would not be possible with a defined variable:

```
DECLARE 1 X UNION,
          2 FLOAT_NUM FLOAT BINARY (24),
          2 BREAKDOWN,
            3 FRAC_1 BIT (7),
            3 EXPONENT BIT (8),
            3 SIGN BIT (1),
            3 FRAC_2 BIT (16);
```

The union X has two immediate members, FLOAT_NUM (a floating-point variable) and BREAKDOWN. The members of BREAKDOWN are bit-string variables that overlay the storage occupied by FLOAT_NUM and provide access to the individual components of its internal representation. Assignment to FLOAT_NUM modifies the members of BREAKDOWN, and vice versa. For example:

```
EXPONENT = '0'B;
SIGN = '1'B;

FLOAT_NUM = FLOAT_NUM + 1;
```

The first two assignment statements set the exponent and sign fields of FLOAT_NUM to the reserved operand combination; the expression FLOAT_NUM + 1 causes a reserved operand exception to occur.

Note that, unlike the character-string example which precedes it, the example above depends on the VAX internal representation of data.

10.2.3 Structure-Qualified References

To refer to a structure in a program, you use the major structure name, minor structure names, and individual member names. Member names need not be unique even within the same structure. To refer to the name of a member or minor structure, you must ensure only that the reference uniquely identifies it. You can qualify the variable name by preceding it

with the name(s) of higher-level (lower-numbered) variable(s) in the structure; names in this format, called a qualified reference, must be separated by periods (.).

The following sample structure definition illustrates the rules for identifying names of variables within structures:

```
DECLARE 1 STATE,
          2 NAME CHARACTER (20),
          2 POPULATION FIXED (10),
          2 CAPITAL,
            3 NAME CHARACTER (30),
            3 POPULATION FIXED (10,0),
          2 SYMBOLS,
            3 FLOWER CHARACTER (20),
            3 BIRD CHARACTER (20);
```

The rules for selecting and specifying variable names for structures are as follows:

- The name of the major structure is subject to the rules for the scope of variables in a program.
- The name of any minor structure or member in a structure can be qualified by the names of higher-level members in the structure. The variable names must be specified from left to right in order of increasing level numbers, separated by periods. The members of the above sample, completely qualified, are

  STATE.NAME
  STATE.POPULATION
  STATE.CAPITAL.POPULATION
  STATE.CAPITAL.NAME
  STATE.SYMBOLS.FLOWER
  STATE.SYMBOLS.BIRD

- Names of minor structures or members within structures do not have to be qualified if they are unique within the scope of the name. The following names in the sample structure can be referred to without qualification (so long as there are no other variables with these names):

  CAPITAL
  SYMBOLS
  FLOWER
  BIRD

- Intermediate qualification names can be omitted if the reference remains unambiguous. The following references to members in the sample structure are valid:

  STATE.FLOWER
  STATE.BIRD

If a name is ambiguous, the compiler cannot resolve the reference and issues a message. In the example, the names POPULATION and NAME are ambiguous.

You can specify the name of a major or minor structure in an assignment statement only if the source expression and the target variable are identical in size and structure, and all corresponding members have the same data types.

10.2.4 Arrays of Structures

An array of structures is an array whose elements are structures. Each structure has identical logical levels, minor structure names, and member names and attributes.

For example, a structure STATE can be declared an array:

```
DECLARE 1 STATE (50),
          2 NAME CHARACTER (20) VARYING,
          2 POPULATION FIXED (31),
          2 CAPITAL,
            3 NAME CHARACTER (30) VARYING,
            3 POPULATION FIXED (31),
          2 SYMBOLS,
            3 FLOWER CHARACTER (20),
            3 BIRD CHARACTER (20);
```

A member of a structure that is an array inherits the dimensions of the structure. For example, the member CAPITAL.NAME of the structure STATE inherits the dimension 50. You must use a subscript whenever you refer to the variable CAPITAL.NAME, as in this example:

```
PUT LIST (CAPITAL.NAME(I)) ;
```

A subscript for a member of a structure that is an array element can appear following any name within a qualified reference. For example, all of these references are equivalent:

```
STATE(10).CAPITAL.NAME
STATE.CAPITAL(10).NAME
STATE.CAPITAL.NAME(10)
```

10.2.4.1 Arrays of Structures That Contain Arrays

A structure that is defined with a dimension can have members that are arrays. For example:

```
DECLARE 1 STATE (50),
          2 AVERAGE_TEMPS(12) FIXED DECIMAL (5,2),
              .
              .
              .
```

In this example, the elements of the array STATE are structures. At the second level of the hierarchy of each structure, AVERAGE_TEMPS is an array of 12 elements. Because AVERAGE_TEMPS inherits the dimension of STATE, any of AVERAGE_TEMPS's elements must be referred to by two subscripts:

1. The first subscript references an element in STATE.
2. The second subscript references an element in AVERAGE_TEMPS.

These subscripts can appear following any name in the qualified reference. For example:

```
STATE(3).AVERAGE_TEMPS(4)
STATE.AVERAGE_TEMPS(3,4)
```

These references are equivalent.

Note the following rules for specifying subscripts for members of structures containing arrays:

- The number of subscripts specified for any member must include any dimensions inherited from a major or minor structure declaration, as well as those specified for the member itself.
- The subscripts that refer to a member of a structure in an array do not have to follow immediately the name to which they apply. However, the order of subscripts must be preserved.
- The total number of dimensions, including the inherited dimensions, must not exceed eight.

10.2.4.2 Connected and Unconnected Arrays

A connected array is one whose elements occupy consecutive locations in storage. For example:

```
DECLARE NEWSPAPERS (10) CHARACTER (30);
```

In storage, the 10 elements of the array NEWSPAPERS occupy 10 consecutive 30-byte units. Thus, NEWSPAPERS is a connected array.

A connected array is valid as the target of an assignment statement, as long as the source expression is a similarly dimensioned array or a single scalar value.

In an unconnected array, the elements do not occupy consecutive storage locations. An unconnected array is not valid in an assignment statement or as the source or target of a record I/O statement. A structure with the

dimension attribute always results in unconnected arrays. When a structure is dimensioned, each member of the structure inherits the dimensions of the structure and becomes, in effect, an array. For example:

```
DECLARE 1 STATE (50),
          2 NAME CHARACTER (20) VARYING,
          2 POPULATION FIXED (31);
```

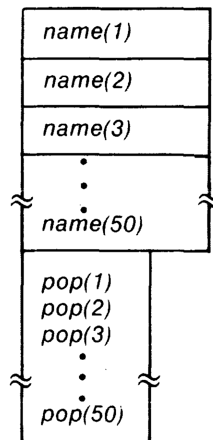
The members NAME and POPULATION of the major structure STATE inherit the dimension 50 from the major structure. When PL/I allocates storage for a structure or a dimensioned structure, each member is allocated consecutive storage locations; thus the elements of the arrays NAME and POPULATION are not connected.

Figure 10–1 illustrates the storage of connected and unconnected arrays.

CONNECTED:

```
DECLARE 1 STATE,
          2 NAME (50) CHAR(20),
          2 POP (50) FIXED(10);
```

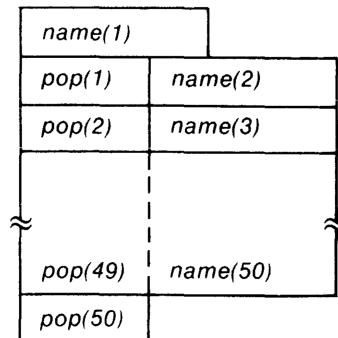
The members NAME and POP of the structure STATE are dimensioned. The elements of each array occupy consecutive storage locations.



UNCONNECTED:

```
DECLARE 1 STATE (50),
          2 NAME CHAR(20),
          2 POP FIXED(10):
```

The array STATE is dimensioned. Its members NAME and POP inherit the dimension: each of these variables is an array of 50 elements, but the elements do not occupy consecutive storage locations.



ZK-028-81

Figure 10–1: Connected and Unconnected Arrays

Chapter 11

Declarations

Before you can use a variable in a PL/I program, you must declare it with the DECLARE statement. When you declare a variable, you give it one of the fundamental data types described in Chapter 8; you may assign it to one of the storage classes described in Chapter 9; and you may make it an array or structure variable, as described in Chapter 10.

This chapter covers two topics:

- Section 11.1 describes the syntax of the DECLARE statement.
- Section 11.2 describes the scope of a declaration, that is, the region of a program in which a variable is known.

11.1 DECLARE Statement

The DECLARE statement specifies the attributes associated with names. Its general format is

```
{ DECLARE }  declaration[,...];
{ DCL     }
```

declaration

One or more declarations consisting of an identifier and attributes. Formally, each declaration has the format

```
[level] identifier [(bound-pair,...)] [attribute ...]
```

or

```
[level] (declaration,...) [(bound-pair,...)] [attribute ...]
```

Bound pairs are used to specify the dimensions of arrays. If bound pairs are present, they must be in parentheses and must immediately follow the identifier or the parenthetical list of declarations. (Section 10.1 describes arrays.)

Levels are used to specify the relationship of members of structures; if a level is present in the declaration, it must be written first. (Section 10.2 describes structures.)

The format of the DECLARE statement varies according to the number and nature of the items being declared. It can list a single identifier, optionally specifying a level, bound-pair list, and other attributes for that identifier. Alternatively, the statement can include, in parentheses, a list of declarations to which the level and all subsequent attributes apply. The declarations in the second case can be simple identifiers or can include attributes that are specific to individual identifiers. The various formats are described individually, below.

11.1.1 Simple Declarations

A simple declaration defines a single name and describes its attributes. Its format is

```
DECLARE identifier [attribute ...] ;
```

identifier

A 1- to 31-character user-supplied name, which must be unique within the current block. An identifier can consist of any of the alphanumeric characters A through Z, a through z, 0 through 9, $ and _, but must begin with an alphabetic letter, dollar sign, or underscore. Note that PL/I does not distinguish between upper- and lowercase letters in a declaration.

attribute ...

One or more attributes of the name. Attributes, if specified, must be separated by spaces. They can appear in any order. The valid attribute keywords and their meanings are described with their data types in Chapters 8 and 9.

Some examples of simple declarations are

```
DECLARE COUNTER FIXED BINARY (7);
DECLARE TEXT_STRING CHARACTER (80) VARYING;
DECLARE INFILE FILE;
```

Names that are not given specific attributes in a DECLARE statement or that are referenced without being declared are given the default attributes

```
BINARY FIXED (31) AUTOMATIC
```

The compiler issues a warning message whenever it gives a name to these default attributes.

11.1.2 Multiple Declarations

Multiple declarations define two or more names and their individual attributes. This format of the DECLARE statement is

```
DECLARE identifier [attribute ...]
          [,identifier [attribute ...]] ...;
```

When you specify more than one set of names and their attributes, separate each name and attribute set from the preceding set with a comma. A semicolon must follow the last name.

An example of a multiple declaration is

```
DECLARE COUNTER FIXED BINARY (7),
        TEXT_STRING CHARACTER (80) VARYING,
        Y FILE;
```

This DECLARE statement defines the variables COUNTER, TEXT_STRING, and Y. The attributes for each variable follow its name.

11.1.3 Factored Declarations

When two or more names have the same attribute(s), you can combine the declarations into a single factored declaration. This format of the DECLARE statement is

```
DECLARE (identifier[,identifier,...])
          [attribute ...];
```

When you use this format, you must place names that share common attributes within parentheses, separated by commas. The attributes that follow the parenthetical list are applied to all the named identifiers.

Some examples of factored declarations are

```
DECLARE (COUNTER, RATE, INDEX) FIXED BINARY (7);
DECLARE (INPUT_MESSAGE, OUTPUT_MESSAGE, PROMPT)
        CHARACTER (80) VARYING;
```

The variables COUNTER, RATE, and INDEX share the attributes FIXED BINARY (7). The variables INPUT_MESSAGE, OUTPUT_MESSAGE, and PROMPT share the attributes CHARACTER (80) VARYING.

You can also specify, within the parentheses, attributes that are unique to individual variable names, using this format:

```
DECLARE (identifier attribute ...,
          identifier [attribute ...],...)
          attribute ...
```

For example:

```
DECLARE (INFILE INPUT RECORD,
         OUTFILE OUTPUT STREAM) FILE;
```

The DECLARE statement declares INFILE as a RECORD INPUT file and OUTFILE as an OUTPUT STREAM file.

The parentheses can be nested. For example:

```
DECLARE ( (INFILE INPUT, OUTFILE OUTPUT) RECORD,
        SYSFILE STREAM ) FILE;
```

The DECLARE statement declares INFILE as a RECORD INPUT file, OUTFILE as a RECORD OUTPUT file, and SYSFILE as a STREAM INPUT file (STREAM implies INPUT).

11.1.4 Declarations Outside of Procedures

A variable may be declared outside of any procedure. Any variable so declared will be visible within all procedures contained by the module; that is, the scope of the variable will be all procedures in the module. The format for declarations outside of procedures is the same as for other declarations except that variables may have any storage class except AUTOMATIC. If a storage class is not specified, STATIC is supplied.

The following example illustrates the use of this type of declaration:

```
    DECLARE A STATIC FIXED BINARY(31);
        .
        .
        .
FIRST: PROCEDURE;
    DECLARE B FIXED BINARY(31);
        .
        .
        .
END FIRST;

SECOND: PROCEDURE;
    DECLARE C FIXED BINARY(31);
        .
        .
        .
END SECOND;
```

In this example, variable A is visible in both the FIRST and SECOND procedures, but variables B and C are visible only in their containing procedures.

(Section 11.2 describes the scope of names.)

11.1.5 Initializing Variables in the DECLARE Statement

You can use the INITIAL attribute to provide an initial value for a declared variable. The format of the INITIAL attribute is

INITIAL (initial-element[,initial-element...])

initial-element

A construct that supplies a value for the initialized variable. The value must be valid for assignment to the initialized variable. If the initialized variable is an array, a list of initial elements separated by commas is used to initialize individual elements. The number of initial elements must be one for a scalar variable and must not exceed the number of elements of an array variable. Each initial element must be one of the following forms:

- string-constant
- (iteration-factor) (string-constant)
- [(iteration-factor)] arithmetic-constant
- [(iteration-factor)] scalar-reference
- [(iteration-factor)] (scalar-expression)
- [(iteration-factor)] *

The iteration factors are nonnegative integer-valued expressions that specify the number of successive array elements to be initialized with the following value. (Notice that a string constant must be in parentheses if it is used with an iteration factor.)

The asterisk form specifies that the corresponding array elements are to be skipped during the initialization.

Some examples are

```
DECLARE RATE FIXED DECIMAL (2,2) STATIC INITIAL (.04);
DECLARE EOF BIT STATIC INITIAL ('1'B);
DECLARE BELL_CHAR BINARY STATIC INITIAL ('07'B4);
DECLARE OUTPUT_MESSAGE CHARACTER(20) STATIC
            INITIAL ('GOOD MORNING');
DECLARE (A INITIAL ('A'), B INITIAL ('B'),
      C INITIAL ('C')) STATIC CHARACTER;
DECLARE QUEUE_END POINTER STATIC INITIAL(NULL());
DECLARE TABLE (30,3) BINARY STATIC INITIAL ( (90) 10 );
```

The last example initializes all elements of the array TABLE with the value 10.

In a factored declaration, a single initial value applies to all the declared variables, not just to the first one. For example:

```
DECLARE (A,B,C) BINARY STATIC INITIAL (10);
```

This statement declares the integers A, B, and C, all with an initial value of 10.

The following restrictions apply to the use of the INITIAL attribute:

- You cannot specify the INITIAL attribute for a structure variable. Instead, initialize individual members of the structure.
- You cannot specify the INITIAL attribute for a variable or member of a variable that has any of these attributes:

BASED	FILE
CONTROLLED	LABEL
DEFINED	parameter
ENTRY	

- You cannot specify the INITIAL attribute for a member of a structure unless the entire structure was declared with the STATIC or AUTOMATIC attribute.
- If the initialized variable is STATIC, only constants and references to the NULL built-in function are allowed. They may be used with a constant iteration factor and may be enclosed in parentheses.
- Variables and functions (except for parameters) occurring in an initial element must not be declared in the same block as the variable being initialized.

You can generally initialize both static and automatic variables. However, you realize a saving in execution time only with static variables. With automatic variables, the initial values are assigned at block activation, so there is little or no difference in execution time between use of the INITIAL attribute and assignment statements.

11.2 Scope of Declarations

The scope of a declaration of a name is that region of the program in which the name is known. A declaration of a name is known in

- The block in which it is declared.
- Any blocks contained within the declaring block, so long as the name is not redeclared in the contained block.
- Any procedures contained in the program, if the name is declared outside of a procedure.

Two or more declarations of the same name are not allowed in a single block (unless one or more of the declarations are of structure members). Two declarations of the same name in different blocks denote distinct objects unless both specify the EXTERNAL attribute. All EXTERNAL declarations of a particular name denote the same variable or constant,

and all must agree as to its properties. Note that EXTERNAL is supplied by default for declarations of ENTRY and FILE constants. It must be specified explicitly for variables.

Figure 11-1 illustrates the scope of internal names.

```
DECLARE Z STATIC FIXED;

MAINP: PROCEDURE OPTIONS (MAIN);

DECLARE (X, Y, VALUE) FIXED;
|
|   ALPHA: PROCEDURE;
|   |
|   |   BETA: BEGIN;
|   |   DECLARE VALUE FLOAT;
|   |   |
|   |   |   GOTO ERROR;
|   |   |
|   |   END BETA;
|   |
|   |   ERROR:
|   END ALPHA;
|
|   CALC: PROCEDURE;
|   DECLARE (SUM,TOTAL) FLOAT;
|   |
|   END CALC;
END MAINP;
```

Name	Scope
Z	MAINP, ALPHA, BETA, and CALC
MAINP	MAINP, ALPHA, BETA, and CALC
X, Y	MAINP, ALPHA, BETA, and CALC
VALUE (MAINP)	MAINP, ALPHA, and CALC
ALPHA	MAINP, BETA, and CALC
BETA	ALPHA
VALUE (BETA)	BETA
ERROR	ALPHA, BETA
CALC	MAINP, ALPHA
SUM, TOTAL	CALC

ZK-1257-83

Figure 11-1: Scope of Internal Names

Declarations may appear outside of procedures and, if contained within the same block, have meaning throughout all procedures contained in the block. However, if there are multiple blocks, declarations outside of procedures must have the EXTERNAL attribute if they are to be recognized by all blocks and procedures in the program.

For example:

```
   DECLARE X FIXED EXTERNAL STATIC;

A: PROCEDURE OPTIONS(MAIN);

   DECLARE B ENTRY;
       .
       .
       .
   END A;

B: PROCEDURE;
    .
    .
    .
   END B;
```

In this example, the variable X has meaning in both procedures. Since they incorporate two different files, X must be declared with the EXTERNAL attribute. If X is declared with the INTERNAL attribute, X is recognized only in the first procedure.

Chapter 12

Expressions and Assignments

An expression is a representation of a value or of the computation of a value, and an assignment gives the value contained in an expression to a variable. Together, expressions and assignments form the mechanism for performing computation.

This chapter describes the following topics:

- The assignment statement
- Operators and operands, the elements of an expression
- The manner in which expression evaluation takes place, including the sequence and precedence of operations performed
- Conversion of the data types of operands during expression evaluation and assignment
- Pseudovariables, which can be assigned values in assignment statements

12.1 Assignment Statement

The assignment statement gives a value to a specified variable. Its format is

```
target = expression;
```

target

The name of the variable to be assigned a value. It can be

- Any reference to a scalar variable or scalar array element.
- A pseudovariable (for example, SUBSTR).
- A reference to a major or minor structure name or any member of a structure.
- A reference to an array variable.

expression

Any valid expression.

PL/I evaluates an assignment statement and performs the assignment as follows:

1. The target is evaluated. If it contains a pseudovariable, any expressions in the argument list are evaluated. (Section 12.5 describes pseudovariables.)
2. The expression on the right-hand side of the assignment statement is evaluated, producing a result. An expression can consist of many subexpressions and operations, each of which will be evaluated. Section 12.3 describes expression evaluation.
3. If the data type of the result does not match that of the target variable, the resulting value is converted to the data type of the target. Such an implicit conversion may produce a warning message from the compiler.

Some general rules regarding the types of data you can specify in assignment statements are given below. For more complete information about data conversion in assignments, see Section 12.4.3 and Appendix A.

- Area data—You can specify an area variable as the target of an assignment statement only in the following case:

  ```
  area-variable-1 = area-variable-2;
  ```

  where both areas have the same extent. The complete contents of the source are copied to the target.

- Arithmetic data—PL/I converts an arithmetic expression to the type of the target, if their types are different. If the target is a character- or bit-string variable, PL/I converts the arithmetic expression to its character- or bit-string equivalent.

  A character-string expression can be converted to the data type of an arithmetic target only if the string consists solely of characters that have numeric equivalents.

- Arrays—You can specify an array variable as the target of an assignment statement in only the following ways:

  - array-variable = expression;

    where expression yields a scalar value. Every element of the array is assigned the resulting value.

  - array-variable-1 = array-variable-2;

    where the specified array variables have identical data type attributes and dimensions. Each element in array-variable-1 is assigned

the value of the corresponding element in array-variable-2. The storage occupied by the two arrays must not overlap.

Any array variable specified in an assignment statement must occupy connected storage.

All other specifications of an array variable as the target of an assignment statement are invalid.

- Bit data—When the target of an assignment is a bit-string variable, the resulting expression is converted to bit, if necessary, and truncated or padded with trailing zeros to match the length of the target. (Section 8.4 describes the effects of this process.)

- Character data—When the target of an assignment is a fixed-length character string, the resulting expression is converted to character, if necessary, and truncated on the right or padded with trailing spaces to match the length of the target. If the target is a varying-length character string, the resulting expression is truncated on the right if it exceeds the maximum length of the target.

  When one character-string variable is assigned to another, the storage occupied by the two variables cannot overlap.

- Entry data—If the specified expression is an entry constant, an entry variable, or a function reference that returns an entry value, the target must be an entry variable.

- Label data—If the specified expression is a label constant, a label variable, or a function reference that returns a label value, the target must be a label variable.

- Pointer and offset data—If the specified expression is a pointer or offset, or a function reference that returns either, the target must be a pointer or offset variable.

- Structures—You can specify the name of a major or minor structure as the target of an assignment statement only if the source expression is an identical structure with members in the same hierarchy and with identical sizes and data type attributes. The storage occupied by the two structures must not overlap.

  Any structure variable specified in an assignment statement must occupy connected storage (see Section 10.2.4).

12.2 Operators and Operands

An operator is a symbol that requests a unique operation. Operands are the expressions on which operations are performed.

12.2.1 Operators

A prefix operator precedes a single operand. The prefix operators are the unary plus (+), the unary minus (-), and the logical not (^).

- The plus sign can prefix an arithmetic value or variable. However, it does not change the sign of the operand.
- A minus sign reverses the sign of an arithmetic operand.
- The ^ prefix operator performs a logical NOT operation on a bit-string operand; the bit value is complemented.

Some examples of expressions containing prefix operators are

```
A = +55;
B = -88;
BITC = ^BITB;
```

An infix operator appears between two operands, and indicates the operation to be performed on them. PL/I has infix operators for arithmetic, logical, and relational (comparison) operations, and for string concatenations. Some examples of expressions containing infix operators are

```
RESULT = A / B;
IF NAME = FIRST_NAME || LAST_NAME THEN GOTO NAMEOK;
```

An expression can contain both prefix and infix operators, for example:

```
A = -55 * +88;
```

Prefix and infix operators can be applied to expressions by using parentheses for grouping.

The categories of operator and the operator symbols are listed in Table 12-1.

Table 12-1: Operators

Category	Symbol	Operation
Arithmetic operators	+	Addition or prefix plus
	-	Subtraction or prefix minus
	/	Division
	*	Multiplication
	**	Exponentiation
Relational (or comparison) operators	>	Greater than
	<	Less than
	=	Equal to
	^>	Not greater than
	^<	Not less than
	^=	Not equal to
	>=	Greater than or equal to
	<=	Less than or equal to

Table 12-1 (Cont.): Operators

Category	Symbol	Operation
Bit-string (or logical) operators	^	Logical NOT
	&	Logical AND
	¦ or !	Logical OR
Concatenation operator	¦¦ or !!	String concatenation

12.2.2 Operands

Since all operators must yield scalar values, operands may not be arrays or structures. The data type that you can use for an operand in a specific operation depends on the operator:

- Arithmetic operators must have arithmetic operands; if the operands are of different arithmetic types, they are converted before the operation to a single type, called the derived data type. Section 12.4.1 describes this process.
- Logical operators must have bit-string operands.
- Relational operators must have two operands of the same type. (Note, however, that comparisons are allowed between offsets and pointers.)
- The operators greater than (>), less than (<), not greater than (^>), not less than (^<), greater than or equal to (>=), and less than or equal to (<=) are valid only with computational operands.
- The concatenation operator must have two bit-string operands or two character-string operands.

12.3 Expression Evaluation and Precedence of Operations

In a PL/I program, you can use expressions to

- Indicate constant values or scalar variables. For example:

```
A = 55;
NAME = 'HECTOR';
B = A;
```

- Perform algebraic or logical calculations on variables or constants. For example:

```
B = A + 10;
C = A + B * 40;
B = ^A;
COMMON - A & B;
```

- Compare the values of two or more expressions and obtain a Boolean result. For example:

```
IF A < B THEN C = 10;
IF NAME = SAVED_NAME THEN GOTO REPEAT;
```

- Concatenate character- or bit-string values. For example:

```
NAME = FIRST_NAME¦¦LAST_NAME;
```

All expressions except simple constants and references consist of an operator and one or more operands. Each operator requires operands of specific types (either arithmetic, character-string, or bit-string) and produces a result of a specific type. The operands may be constants, variable references, function references, or other expressions, as long as they are objects of the type required by the operator.

Built-in functions may also be considered operators in this sense, and their arguments, operands.

All VAX-11 PL/I expressions and functions have scalar results.

Expressions are evaluated from left to right, with the following qualifications:

- Some PL/I operators take precedence over others used in the same expression. Operations with higher priority are evaluated first, and their results are used as single operands. The rules of precedence usually guarantee an algebraically correct result without the use of parentheses.

  All built-in functions are of equal priority.

  Table 12-2 lists the priorities of PL/I operators. In Table 12-2, low numbers indicate high priority; that is, the exponentiation operator (**) has the highest priority and the OR operator (¦), the lowest.

Table 12-2: Precedence of Operations

Operator	Priority	Operator	Priority
**	1	>	5
+ (prefix)	1	<	5
- (prefix)	1	^>	5
^	1	^<	5
*	2	=, ^=	5
/	2	<=	5
+ (infix)	3	>=	5
- (infix)	3	&	6
¦¦	4	¦	7

- You can enclose any expression in parentheses to override the usual rules of precedence. Expressions at the deepest level of nested parentheses are always evaluated first, and their results used as single operands.
- Exponential operations of the form A**B**C are evaluated from right to left.
- The run-time evaluation of a logical expression may be terminated as soon as its result is known. For instance, evaluation of the expression

  ```
  A & USER_FUNCTION(ALPHA,BETA)
  ```

  may be terminated without evaluating the USER_FUNCTION reference if the evaluation of A results in a "false" Boolean value.
- You cannot count on left-to-right evaluation of an expression, since the compiler may evaluate subexpressions in any order that produces an algebraically correct result. For example, in the expression

  ```
  A+B+FUNC(I)+C
  ```

  you cannot assume that the subexpression A+B will be evaluated and the result stored before FUNC(I) is evaluated. In other words, if FUNC(I) alters A, B, or C, results may not be as expected.
- If a function referenced in an expression executes a nonlocal GOTO statement, the expression is not evaluated further.

12.4 Conversion of Operands and Expressions

Data conversion in PL/I takes place in many contexts, not all of them obvious ones. Program results that seem improper may in fact be caused by data conversion at some point in the program's execution. Section 12.4.1 describes how arithmetic operands of different types are converted to a single derived type during expression evaluation. Section 12.4.2 describes how you can control conversions precisely by using conversion built-in functions designed for that purpose. Section 12.4.3 describes several contexts in which VAX-11 PL/I automatically converts data from one type to another—for example, in input and output by the GET and PUT statements. (Appendix A contains precise rules for these conversions.)

12.4.1 Derived Data Types for Arithmetic Operations

Even though arithmetic operands can be of different arithmetic types, all operations will be performed on objects of the same type. Any set of

operands of different arithmetic types has an associated derived type, as follows:

- If any operand has the attribute BINARY, the derived base is BINARY. Otherwise, the derived base is DECIMAL.
- If any operand has the attribute FLOAT, the derived scale is FLOAT. Otherwise, the derived scale is FIXED.

All arithmetic operations except exponentiation are performed in the derived type of the two operands. Exponential operations are performed in a data type that is based on the derived type of the operands. All operations, including exponentiation, have results of the same type as that in which they are performed.

The result of an arithmetic operation may be assigned to a target variable of any computational type. The result is converted to the target type, following the rules in Section 12.4.3 and Appendix A. Such conversion may, however, result in a warning message from the compiler.

12.4.2 Built-In Conversion Functions

The built-in conversion functions can take arguments that are either arithmetic or string expressions. They are, in fact, often used to convert an operand to the type required in a certain context—for instance, to convert a bit string to an arithmetic value for use as an arithmetic operand.

For the purpose of these functions, and in a few other contexts, derived arithmetic attributes are also defined for bit- and character-string expressions:

- The derived type of a bit string is fixed-point binary; its converted precision is 31, with a scale factor of 0.
- The derived type of a character string is fixed-point decimal; its converted precision is also 31, with a scale factor of 0.

PL/I uses these derived attributes to determine the precision of values returned by the conversion functions if no precision is specified in the functions' argument lists. Of course, the value of a string argument must also be convertible to the result type; for instance, '1.333' is convertible to arithmetic, but 'ABCD' is not.

Table 12–3 indicates which built-in functions you should use for each conversion between an arithmetic and a nonarithmetic type. In addition, you can use the BINARY, DECIMAL, FIXED, and FLOAT built-in conversion functions to control conversions between two arithmetic types.

Table 12-3: Built-In Functions for Conversions Between Arithmetic and Nonarithmetic Types

Conversion	Function
Arithmetic to Bit	BIT(s[,l])
Arithmetic to Character	CHARACTER(s[,l])
Bit to Arithmetic	BINARY(x[,p[,q]])
Bit to Character	CHARACTER(s[,l])
Character to Bit	BIT(s[,l])
Character to Decimal	DECIMAL(x[,p[,q]])
Character to Float	FLOAT(x,p)
Character to Integer	BINARY(x[,p[,q]])

For more information, see the individual descriptions of the built-in functions in Section 19.2.

12.4.3 Implicit Conversion During Assignment

During assignment, VAX-11 PL/I automatically converts the derived data type of an expression to the data type of a target, if necessary. In assignments, conversions are defined between the noncomputational types POINTER and OFFSET, and between any two computational types. The rules for assignments apply to

- Assignment statements.
- Arguments passed to a procedure.
- Values specified in a RETURN statement.
- An argument converted by the built-in function FIXED, FLOAT, BINARY, DECIMAL, BIT, or CHARACTER.
- Conversions to and from character strings performed by the PUT and GET statements, respectively.

However, a conversion during assignment results in an error if PL/I cannot perform it in a meaningful way. For example, you can assign the string '123.4' to a fixed decimal variable; you cannot, however, assign the string 'ABCD' to the same variable. Similarly, an assignment of an arithmetic type to a fixed variable results in the FIXEDOVERFLOW condition if integral digits are lost.

Although VAX-11 PL/I performs conversions in assignment statements, such conversions may represent programming errors and are, furthermore, in violation of the PL/I G subset standard. Therefore, the compiler issues a warning message that an implicit conversion is taking place. These messages do not terminate the compilation and may not indicate errors; they simply alert you to the fact that your program converts one data type to another in a way that may cause a problem when the program is run. You can prevent such warning messages in two ways:

- Use the /NOWARNINGS qualifier to the PLI command to suppress diagnostic warning messages. (The compiler will continue to print messages of greater severity.) However, you run the risk of missing important diagnostic information.
- Use the built-in conversion functions to convert data types explicitly. This method is recommended. Section 12.4.2 summarizes the functions.

For example:

```
DECLARE (A,B) FIXED DECIMAL (5,2);
    A = '123.45';                  /* Warning message */
    B = FIXED('123.45',5,2);       /* No warning */
```

Both assignment statements assign the same value to their targets; however, the first statement causes a warning message from the compiler, while the second statement does not.

Appendix A defines the rules and results of the following types of conversion:

- Assignments to arithmetic variables
- Assignments to bit-string variables
- Assignments to character-string variables
- Assignments to pictured variables
- Conversions between offsets and pointers

12.5 Pseudovariables

In VAX-11 PL/I, the pseudovariables PAGENO, STRING, SUBSTR, and UNSPEC can substitute, in certain assignment contexts, for an ordinary variable reference. For example:

```
SUBSTR(S,2,1) = 'A';
```

assigns the character 'A' to a 1-character substring of S, beginning at the second character of S.

You can use a pseudovariable wherever the following three conditions are true:

1. The syntax specifies a variable reference.
2. The context is one that explicitly assigns a value to the variable.
3. The context does not require the variable to be addressable.

The principal contexts in which pseudovariables are used are

- On the left side of an assignment statement.
- As the input target of a GET statement.

Note that you cannot include a pseudovariable in an argument list. For example:

```
CALL P(SUBSTR(S,2,1));
```

Here, SUBSTR is interpreted as a built-in function reference, not as a pseudovariable. The actual argument passed to procedure P is a dummy argument containing the second character of string S.

The VAX-11 PL/I pseudovariables are described individually in alphabetic order in the following paragraphs.

INT Pseudovariable

The INT pseudovariable assigns a signed integer value to the storage specified. Its format is

INT(reference[,position[,length]]) = expression;

reference

A reference to connected storage. If position and length are not specified, the length of the referenced storage must not exceed 32 bits.

position

A positive integer value that denotes the position of the first bit in the field. If omitted, position defaults to 1, signifying the first bit of the storage denoted by reference. If specified, position must satisfy the condition

$1 \leq \text{position} \leq \text{size(reference)}$

where size(reference) is the length in bits of the storage denoted by reference. A position equal to size(reference) implies a zero-length field.

length

An integer value in the range 0 to 32 that specifies the length of the field. If omitted, length is the number of bits from the bit denoted by

position through the end of the storage denoted by reference. If specified, length must satisfy the condition

$0 \leq \text{length} \leq \text{size(reference)} - \text{position}$

where size(reference) is the length in bits of the storage denoted by reference.

The INT pseudovariable is valid only in an assignment statement. It cannot be used as the target of an input statement or in other instances where pseudovariables are normally acceptable.

The expression to be assigned to the pseudovariable is first converted to the data type FIXED BINARY (31); then, the internal representation of the resulting integer value is assigned to the storage specified by the arguments to INT. If the representation of the value is too large for assignment to the storage, the most significant bits of the integer are removed, and no error is signaled.

For example:

```
DECLARE F FLOAT INITIAL (123.45);

     INT(F,8,8) = 25;       /* Alter the exponent */
     PUT SKIP LIST (F);     /* New value */
```

In this example, the INT pseudovariable is used to modify the exponent field of a floating-point variable. This example prints the value

```
9.5102418E-32
```

Proper interpretation of this result requires understanding of the internal representation of floating-point numbers.

The next example demonstrates how the INT pseudovariable treats cases in which the value is too large for the specified storage:

```
INTOVER: PROCEDURE OPTIONS (MAIN);

DECLARE I15 FIXED BINARY (15),
        I31 FIXED BINARY (31);

ON FIXEDOVERFLOW PUT SKIP LIST ('FIXEDOVERFLOW signaled');

    I31 = -876543;          /* Too big for I15 */

    I15 = I31;              /* Arithmetic assignment */
    INT(I15) = I31;         /* No error signaled */
    PUT SKIP LIST (I15);

    END;
```

This example produces the following output:

```
FIXEDOVERFLOW signaled
   -24575
```

The arithmetic assignment to I15 signals FIXEDOVERFLOW because the value of I31 is outside the range of a FIXED BINARY (15) variable. However, the assignment using the INT pseudovariable does not signal an error; it just copies the low-order 16 bits of the value of I31 into I15's storage.

PAGENO Pseudovariable

The PAGENO pseudovariable refers to the page number of the referenced print file. Assignment to the pseudovariable modifies the current page number. The format (in an assignment statement) is

```
PAGENO(reference) = expression;
```

reference

A reference to an open print file for which the page number is to be set.

PAGENO(reference) is a FIXED BINARY(15) variable; however, values assigned to it must not be negative.

POSINT Pseudovariable

The POSINT pseudovariable assigns an integer value to specified storage. Its format (in an assignment statement) is

```
POSINT(expression[,position[,length]]) = expression;
```

expression

A reference to connected storage. If position and length are not specified, the length of the referenced storage must not exceed 32 bits.

position

A positive integer value that denotes the position of the first bit in the field. If omitted, position defaults to 1, signifying the first bit of the storage denoted by expression. If specified, position must satisfy the condition

$1 \leq \text{position} \leq \text{size(expression)}$

where size(expression) is the length in bits of the storage denoted by expression. A position equal to size(expression) implies a zero-length field.

length

An integer value in the range 0 to 32 that specifies the length of the field. If omitted, length is the number of bits from the bit denoted by position through the end of the storage denoted by expression. If specified, length must satisfy the condition

$0 \leq \text{length} \leq \text{size(expression)} - \text{position}$

where size(expression) is the length in bits of the storage denoted by expression.

The POSINT pseudovariable is valid only in an assignment statement. It cannot be used as the target of an input statement or in other instances where pseudovariables are normally acceptable.

The expression to be assigned to the pseudovariable is first converted to the data type FIXED BINARY (31); then, the internal representation of the resulting integer value is assigned to the storage specified by the arguments to POSINT. If the representation of the value is too large for assignment to the storage, the most significant bits of the integer are removed and no error is signaled.

The POSINT pseudovariable is identical in operation and use to the INT pseudovariable.

STRING Pseudovariable

The STRING pseudovariable interprets a suitable reference as a reference to a fixed-length string. By using it, you can modify an entire aggregate with a single string assignment, or assign a value to a pictured variable as if it were a character-string variable. The format (in an assignment statement) is

```
STRING(reference) = expression;
```

reference

A reference to a variable that is suitable for character-string (or bit-string) overlay defining (see Section 9.5.7.1). The length of the pseudovariable equals the total number of characters (or bits) in the scalar or aggregate denoted by the reference, and must be less than or equal to the maximum length for character-string (or bit-string) data.

Assignment to the STRING pseudovariable modifies the entire storage denoted by the reference.

For example:

```
STRING_PSD_EXAMPLE: PROCEDURE;

DECLARE 1 NAME,
          2 FIRST CHARACTER(10),
          2 MIDDLE_INITIAL CHARACTER(3),
          2 LAST CHARACTER(10);

STRING(NAME)='FRANKLIN D. ROOSEVELT';

        /* NAME.FIRST = 'FRANKLIN D';
           NAME.MIDDLE_INITIAL = '. R';
           NAME.LAST = 'OOSEVELT  '; */
```

```
END STRING_PSD_EXAMPLE;
    .
    .
    .
DECLARE 1 FLAGS,
          2 (A,B,C) BIT(1);
STRING(FLAGS) = '0'B; /* sets all three flags false */
    .
    .
    .
DECLARE P PICTURE 'Z.ZZZV,ZZDB';
GET EDIT (STRING(P)) (A(10));
        /* assigns 10 characters from SYSIN to P,
        without conversion */
```

SUBSTR Pseudovariable

The SUBSTR pseudovariable refers to a substring of a specified string variable reference. Assignment to the pseudovariable modifies only the substring. The format (in an assignment statement) is

SUBSTR(reference,position[,length]) = expression;

reference

A reference to a bit- or character-string variable. If the reference is to a varying-length character string, the substring defined by the position and length arguments must be within the current value of the string. Assignment to the SUBSTR pseudovariable does not change the length of a varying string.

position

An integer expression indicating the position of the first bit or character in the substring. The position must be greater than or equal to one, and less than or equal to LENGTH(reference)+1.

length

An integer expression that indicates the length of the substring. If not specified, length has the value

length = LENGTH(reference) - position + 1

which specifies the substring beginning at the indicated position and ending at the end of the string. The length must satisfy the condition

$0 \leq \text{length} \leq \text{LENGTH(reference)} - \text{position} + 1$

Note that

```
SUBSTR(r,p,l) = v;
```

is equivalent to

```
r = SUBSTR(r,1,p-1)||v||SUBSTR(r,p+l);
```

For example:

```
DECLARE (NAME,NEWNAME) CHARACTER(20) VARYING;

NAME = 'ISAK DINESEN';
NEWNAME = NAME;
SUBSTR(NEWNAME,4) = 'AC NEWTON' ;
/* NEWNAME = 'ISAAC NEWTON' */
```

UNSPEC Pseudovariable

The UNSPEC pseudovariable interprets any reference to a scalar variable as a reference to a bit string. The format (in an assignment statement) is

UNSPEC(reference) = expression;

reference

A reference to a scalar variable. The length of its storage in bits must be less than or equal to the maximum length for bit-string data.

In an assignment of the form

UNSPEC(reference) = value;

the value is converted to a bit string if necessary and copied into the storage of the reference. The value is truncated or zero-extended as necessary to match the length of the storage.

For example:

```
DECLARE X FIXED BINARY (15);

UNSPEC(X) = '110'B;
```

The use of the constant '110'B assigns 3 to X. The two low-order bits of X (that is, X's first two bits of storage) are set; all other bits of X are cleared.

Chapter 13

Procedures

A procedure is the basic executable program unit in PL/I. It consists of a sequence of statements, headed by a PROCEDURE statement and terminated by an END statement, that define an executable set of program instructions.

This chapter describes the following topics:

- Using procedures. Section 13.1 describes the concepts and statements for defining and invoking functions and subroutines and obtaining return values from them.

- External procedures. External procedures are not contained within another block. They can be compiled separately from procedures that invoke them. External procedures can be written in languages other than PL/I. Section 13.2 concentrates on external procedures.

13.1 Using Procedures

This section describes how you define and invoke a procedure, and how the procedure terminates and returns values through its parameter list, its RETURN statement, or both. Section 13.1.1 presents general concepts. Subsequent sections describe the statements and syntax you need to use procedures:

- Section 13.1.2 describes the PROCEDURE statement.

- Section 13.1.3 describes the ENTRY statement, which defines an alternate entry point to a procedure.

- Section 13.1.4 describes the CALL statement, which invokes a procedure as a subroutine.

- Section 13.1.5 describes functions and references to them. A function reference invokes a procedure that returns a single value, which then takes the place of the function reference in the expression.

- Section 13.1.6 describes the RETURN statement, which terminates execution of the current procedure, and (in the case of a function reference) specifies a return value.
- Section 13.1.7 describes the RETURNS attribute and option. The RETURNS attribute is included with a PROCEDURE or ENTRY statement for a function to give the data type returned by the function. The RETURNS option to the ENTRY attribute in the declaration of an external procedure gives the data type returned by the procedure.
- Section 13.1.8 describes methods that PL/I uses to pass arguments to an invoked procedure and return parameters to an invoking procedure. The section also contains the rules that the compiler follows in determining which method to use for a given type of argument.

13.1.1 Procedure Usage Concepts

Two types of procedure can be invoked by another procedure during its execution. They are

- Subroutines, which must be invoked with a CALL statement. Subroutines return values to the invoking procedure only by means of their parameter lists; they must not include an expression in their RETURN statements and must not include a RETURNS option on their PROCEDURE or ENTRY statements.
- Functions, which must be invoked by a function reference. A function reference can appear in place of a scalar value in any appropriate context in a PL/I statement. A function returns to the invoking procedure a single value that becomes the value of the function reference in the invoking procedure. Functions may also return values via their parameter lists. Functions must include a RETURNS option to describe the attributes of the returned value and must specify an expression in their RETURN statements.

Each type of procedure can be passed data or information from the invoking procedure by means of an argument list.

13.1.1.1 Entry Points

The entry points of a procedure are the points at which it can be invoked. The PROCEDURE statement specifies one entry point. Additional entry points may be specified with ENTRY statements within the procedure block. ENTRY statements are allowed anywhere except within a begin block, an ON-unit, or a DO group (except a simple, noniterative DO group).

The labels used on PROCEDURE and ENTRY statements declare those names as entry constants. The scope of the declarations is internal if the

PROCEDURE and ENTRY statements appear in internal procedures, and external if they appear in external procedures.

Note that the declaration of an entry name is made in the block containing the procedure to which the entry point belongs. For example:

```
P: PROCEDURE;

Q: PROCEDURE;
   DECLARE E FIXED BINARY;
   E: ENTRY;
END Q;
```

The entry names E and Q are declared in procedure P. Within procedure Q, E is declared as a fixed-point binary variable. This does not conflict with the declaration of E as an entry in procedure P.

You can invoke an entry point by using the appropriate entry constant as the reference in a CALL statement or function reference. Invoking an entry point enters a procedure at the specified point and activates the procedure block that contains the entry point.

13.1.1.2 Passing Arguments to Subroutines and Functions

You specify arguments for a subroutine or function by enclosing the arguments in parentheses after the procedure or entry point name. Arguments correspond to parameters specified on the PROCEDURE or ENTRY statement of the invoked procedure. For example, you can write a procedure call as follows:

```
CALL COMPUTER (A,B,C);
```

The variables A, B, and C are arguments to be passed to the procedure COMPUTER, which might have a parameter list like this:

```
COMPUTER: PROCEDURE (X, Y, Z);
DECLARE (X,Y,Z) FLOAT;
```

The parameters X, Y, and Z, specified in the PROCEDURE statement for the subroutine COMPUTER, are the parameters of the subroutine. PL/I establishes the equivalence of the arguments A, B, and C with the parameters X, Y, and Z.

For more information about arguments, parameters, and the relationship between them, see Section 13.1.8.

13.1.1.3 Terminating Procedures

You can terminate subroutines and functions in the following ways:

- A RETURN statement—A RETURN statement provides a normal termination for a subroutine or function. For a function, a RETURN statement must specify a return value.

- A STOP statement—A STOP statement normally ends the entire program execution. It does not pass a return value. (The STOP statement signals the FINISH condition, thereby allowing a FINISH ON-unit to execute before the program terminates. For details, see Section 15.1.)
- An END statement—If an END statement closes the procedure block of a subroutine before a RETURN or STOP statement is executed, it has the same effect as RETURN. A function cannot be terminated without a RETURN statement.
- A nonlocal GOTO statement—A GOTO statement that transfers control to a label outside the current block terminates a subroutine or a function. The label specified on the GOTO statement must be known within the block that contains the GOTO statement, and the block containing the specified label must be active when the GOTO statement is executed.

13.1.2 PROCEDURE Statement

The PROCEDURE statement defines the beginning of a procedure block and specifies the parameters, if any, of the procedure. If the procedure is invoked as a function, the PROCEDURE statement also specifies the data type attributes of the value that the function returns to its point of invocation.

The PROCEDURE statement may denote the beginning of an internal or external subroutine or function. Its format is

```
entry-name: { PROCEDURE }  [ (parameter,...) ]
            { PROC      }

                 [ OPTIONS (option,...)            ]
                 [ RECURSIVE                       ] ;
                 [ RETURNS (returns-descriptor)    ]
```

entry-name

A 1- to 31-character identifier denoting the entry label of the procedure. The label cannot be subscripted. The PROCEDURE statement implicitly declares the entry name as an entry constant. The scope of the name is INTERNAL if the procedure is internal, and EXTERNAL if the procedure is external.

parameter,...

One or more parameters, separated by commas, that the procedure expects when it is activated. Each parameter specifies the name of a variable declared in the procedure headed by this PROCEDURE statement. The parameters must correspond one-to-one with arguments specified for the procedure when it is invoked with a CALL statement or in a function reference. See Section 13.1.8 for detailed information about arguments and parameters.

OPTIONS (option,...)

An option that specifies one or more options, separated by commas. The valid options are

IDENT(string)

An option specifying a character-string constant giving the identifying label for the listing and the object module's version for the linker. Only the first 31 characters of the string are placed in the object module.

MAIN

An option specifying that the named procedure is the initial procedure in a program. The identifier of the procedure is the primary entry point for the program. The MAIN option is not allowed on internal procedures, and only one procedure in a program can have the MAIN option.

UNDERFLOW

An option that requests the run-time system to signal underflow conditions when they occur. By default, the run-time system does not signal them.

RECURSIVE

An option that indicates (for program documentation) that the procedure will be invoked recursively: the procedure will be activated while it is currently active. In standard PL/I, the RECURSIVE option must be specified for a procedure to be invoked recursively. However, in VAX-11 PL/I, you may invoke all procedures recursively; the compiler ignores the RECURSIVE option.

RETURNS (returns-descriptor)

An option that specifies that the procedure can be invoked only by a function reference and that specifies the attributes of the function value returned. See Section 13.1.7 for syntax and details.

You must specify RETURNS for functions. It is invalid for procedures invoked by CALL statements.

13.1.3 ENTRY Statement

The ENTRY statement defines an alternate entry point to a procedure. Its format is

```
entry-name: ENTRY [ (parameter,...) ]
                  [ RETURNS (returns-descriptor) ];
```

entry-name

A 1- to 31-character identifier for the entry point. Specifying the entry name implicitly declares the name as an entry constant. The scope of the name is external if the ENTRY statement is contained in an external procedure, and internal if in an internal procedure.

parameter,...

One or more parameters, separated by commas, that the procedure requires at this entry point. Each parameter specifies the name of a variable declared in the block to which this ENTRY statement belongs. The parameters must correspond one-to-one with arguments specified for the procedure when it is invoked via this ENTRY statement. See Section 13.1.8 for detailed information about arguments and parameters.

RETURNS (returns-descriptor)

An option giving, for an entry that is invoked as a function reference, the data type attributes of the function value returned. (See Section 13.1.7 for syntax and details.) For entry points invoked by function references, the RETURNS option is required; for procedures invoked by CALL statements, the RETURNS option is invalid.

An ENTRY statement is not allowed in a begin block, ON-unit, SELECT-group, or DO group except for a simple DO. Additional rules governing the declaration of multiple entry points are

- A particular parameter need not be specified in all of a procedure's entry points (including the point defined by the PROCEDURE statement). However, a reference to the parameter is valid only if the procedure was invoked via one of the entries specifying the parameter.

- In a procedure with multiple entry points, a RETURN statement must be compatible with the entry point by which the procedure was invoked. If the entry point does not have a RETURNS option, the RETURN statement must not specify a return value (and, in addition, the entry point must be invoked as a subroutine—that is, with the CALL statement). If the entry point does have a RETURNS option, the RETURN statement must specify a value that is valid for conversion to the data type specified in the RETURNS option.

- An ENTRY statement is not executable. If control reaches it sequentially, control passes on to the next statement.

The following example shows a procedure with two alternate entry points:

```
QUEUES: PROCEDURE(ELEMENT,QUEUE_HEAD);
     .
     .
     .
```

```
ADD_ELEMENT: ENTRY(ELEMENT);
    .
    .
    .
REMOVE_ELEMENT: ENTRY(ELEMENT);
```

This procedure can be entered by CALL statements that reference QUEUES, ADD_ELEMENT, or REMOVE_ELEMENT. If it is invoked at QUEUES, it must be passed two parameters. At either of the entries ADD_ELEMENT or REMOVE_ELEMENT, it must be passed only one parameter. When it is entered at either alternate entry point, the entire block beginning at QUEUES is activated, but execution begins with the first executable statement following the entry point.

You should avoid unnecessary use of ENTRY statements, because their effect is detrimental to the overall optimization of the program.

13.1.4 CALL Statement

The CALL statement invokes a subroutine. It transfers control to an entry point of a procedure and optionally passes arguments to the procedure. The format of the CALL statement is

```
CALL entry-name [(argument,...)];
```

entry-name

The name of an internal or external procedure that does not have the RETURNS attribute, or the name of an alternate entry point to a procedure. (The entry name can also be an entry variable or a reference to a function that returns an entry value.)

argument,...

The argument list to be passed to the called procedure. If specified, the arguments must correspond to the parameters specified in the PROCEDURE or ENTRY statement that identifies the entry name of the called procedure.

Unless OPTIONS(VARIABLE) is specified in the declaration of an external entry name, the number of arguments must match the number of parameters in the parameter list of the invoked entry name. OPTIONS(VARIABLE) is valid only for use with non-PL/I procedures.

Arguments must be enclosed in parentheses, and multiple arguments separated by commas. See Section 13.1.8 for detailed information about arguments and parameters.

The following example illustrates a main procedure, CALLER, and a call to an internal subroutine, PUT_OUTPUT. PUT_OUTPUT has two parameters, INSTRING and OUTFILE, that correspond to the arguments LINE and DEVICE specified in the CALL statement.

```
CALLER: PROCEDURE OPTIONS(MAIN);
    .
    .
    .
        CALL PUT_OUTPUT(LINE,DEVICE);
    .
    .
    .
PUT_OUTPUT: PROCEDURE(INSTRING,OUTFILE);
    .
    .
    .
END PUT_OUTPUT;
END CALLER;
```

13.1.5 Functions and Function References

A function is a procedure that returns a scalar value. It receives control when its name is referenced in an expression. There are two types of functions:

- PL/I built-in functions
- User-written functions

The built-in functions, which are available in all programs and generally need not be declared, are described in Chapter 19.

A user-written function must

- Contain the RETURNS option on the PROCEDURE statement.
- Specify a value on the RETURN statement. The value must be of a data type that is valid for conversion to the one specified on the RETURNS option.

For example:

```
ADDER: PROCEDURE (X,Y) RETURNS (FLOAT);
DECLARE (X,Y) FLOAT;
        RETURN (X+Y);
        END;
```

The function ADDER has two parameters, X and Y. They are floating-point binary variables declared within the function. When the function is invoked by a function reference, it must be passed two arguments to correspond to these parameters. It returns a floating-point binary value representing the sum of the arguments.

The format of a function reference is

    entry-name ([argument,...])

entry-name

The name of an entry constant or variable used to invoke the function.

argument,...

One or more arguments to be passed to the function. If specified, the arguments must correspond to the parameters specified in the PROCEDURE or ENTRY statement that identifies the entry name of the function.

Arguments must be enclosed in parentheses, and multiple arguments separated by commas. See Section 13.1.8 for detailed information about arguments and parameters.

An argument can be an expression of any data type that is convertible to the data type of the corresponding parameter.

For example, the function ADDER may be referenced as follows:

```
TOTAL = ADDER(5,6);
```

The arguments in the reference to ADDER are converted to FLOAT.

If a function has no parameters, you must specify a null argument list; otherwise, the compiler treats the reference as a reference to an entry constant. Specify a null argument list as follows:

```
GETDATE = TIME_STAMP();
```

This assignment statement contains a reference to the function TIME_STAMP, which has no parameters.

This rule applies to PL/I built-in functions as well; however, if you declare a PL/I built-in function explicitly with the BUILTIN attribute, you need not specify the empty argument list. For example:

```
DECLARE P POINTER,
        NULL BUILTIN;
    .
    .
    .
P = NULL;
```

This example assigns a null pointer value to P. Without the declaration of NULL as a built-in function, the assignment statement would have been

```
P = NULL();
```

13.1.6 RETURN Statement

The RETURN statement terminates execution of the current procedure. Its format is

RETURN [(return-value)] ;

return-value

The value to be returned to the invoking procedure. If the current procedure was invoked by a function reference, a return value must be specified. If the current procedure was invoked by a CALL statement, a return value is invalid.

A return value can be any scalar arithmetic, bit-string, or character-string expression; it can also be an entry, pointer, label, or other noncomputational expression. The return value must be valid for conversion to the data type specified in the RETURNS option of the function.

The action taken by the RETURN statement depends on the context of the procedure activation, as follows:

- If the current procedure is the main, or only, active procedure, the RETURN statement terminates the program. If the RETURN statement specifies a return value, it is returned to the command interpreter, which may then issue a message. See Section 5.2 for details.
- If the current procedure was activated by a CALL statement, control returns to the next executable statement in the calling procedure.
- If the current procedure was activated by a function reference, control returns to continue the evaluation of the statement that contained the function reference.
- If the RETURN statement is executed in a begin block, the effect is to return from the containing procedure.

The RETURN statement must not be immediately contained in an ON-unit or in a begin block that is immediately contained in an ON-unit.

13.1.7 RETURNS Attribute and Option

The RETURNS option must be specified on the PROCEDURE or ENTRY statement if the corresponding entry point is invoked as a function. The RETURNS attribute is specified with the ENTRY attribute, to give the data type of a value returned by an external function. The format of the option or attribute is

```
RETURNS (returns-descriptor)
```

returns-descriptor

One or more attributes that describe the value returned by the function to its point of invocation. The returned value becomes the value of the function reference in the invoking procedure. The attributes must be separated by spaces except for attributes (precision, for example) that are enclosed in parentheses.

The data types you can specify for a returns descriptor are restricted to scalar elements of either computational or noncomputational types. Areas are not allowed.

The extent of a character-string value may be specified as an asterisk (*), to indicate that the string may have any length; in this case, VARYING must not be specified. Otherwise, extents must be specified using unsigned decimal integer constants.

The RETURNS option and RETURNS attribute must not be used for procedures that are invoked by the CALL statement.

The attributes specified in a returns descriptor of a RETURNS attribute must correspond to those specified in the RETURNS option of the PROCEDURE statement or ENTRY statement(s) in the corresponding procedure. For example:

```
CALLER: PROCEDURE OPTIONS (MAIN);
     DECLARE COMPUTER ENTRY (FIXED BINARY)
          RETURNS (FIXED BINARY); /* RETURNS attribute */
     DECLARE TOTAL FIXED BINARY;
          .
          .
          .
     TOTAL = COMPUTER (A+B);
```

The first DECLARE statement declares an entry constant named COMPUTER, which will be used in a function reference to invoke an external procedure. The function reference must supply a fixed-point binary argument. The invoked function returns a fixed-point binary value, which then becomes the value of the function reference.

The function COMPUTER contains

```
COMPUTER:PROCEDURE (X)
RETURNS (FIXED BINARY); /* RETURNS option */
DECLARE (X, VALUE) FIXED BINARY;
    .
    .
    .
RETURN (VALUE);
```

In the PROCEDURE statement, COMPUTER is declared as an external entry constant, and the RETURNS option specifies that the procedure returns a fixed-point binary value to the point of invocation. The RETURN statement specifies that the value of the variable VALUE is returned by COMPUTER. If the data type of the returned value does not match the one specified in the RETURNS option, PL/I converts the value to the correct data type according to the rules given in Section 12.4.3 and Appendix A.

13.1.8 Parameters and Arguments

A parameter is a variable that occurs in the parameter list of a PROCEDURE or ENTRY statement. When the entry point is invoked, each parameter in the list is associated with an argument variable. Within the procedure invocation, any reference to the parameter is equivalent to a reference to the associated argument variable.

If the invoked entry point is external to the invoking procedure, the attributes of the parameters must be described in parameter descriptors, which are part of the declaration of the external entry point.

Each entry point in a procedure must have a parameter list if that entry point is to be invoked with an argument list. Multiple entry points in a procedure do not need to have identical parameters, but a reference to a parameter is valid only if the procedure was invoked via an entry point that specified that parameter.

An argument is an expression or variable reference denoting a value to be passed to the invoked procedure. A procedure must be invoked with the same number of arguments as it has parameters; the maximum number is 253. The argument variable associated with a parameter, or "actual argument," may be a variable written in the argument list or a dummy argument. The compiler creates a dummy argument when the specified argument is a constant or expression existing only for the duration of the procedure invocation. Therefore, references in the invoked procedure to the parameter associated with a dummy argument do not modify any storage in the invoking procedure.

An argument list consists of zero or more arguments specified in the invocation of a procedure, built-in function, or built-in subroutine. In the case of built-in functions, arguments are expressions that supply values to the built-in function, and the argument types must be those required by it. In the case of user-defined procedures, arguments correspond to parameters defined on the PROCEDURE or ENTRY statement of the invoked procedure.

13.1.8.1 Rules for Specifying Parameters

The general rules listed below for specifying parameters are followed by specific rules that pertain only to certain data types.

- A parameter must be declared explicitly in a DECLARE statement (to give it a data type) within the invoked procedure. This declaration must not be part of a structure.

- A parameter must not be declared with any of these attributes:

  AUTOMATIC GLOBALREF
  DEFINED INITIAL

CONTROLLED READONLY
EXTERNAL STATIC
GLOBALDEF VALUE

- A maximum of 253 parameters can be specified for an entry point.
- The parameters of an external entry must be explicitly specified by parameter descriptors in the declaration of the entry constant. The parameters of a procedure that is invoked via an ENTRY variable must be specified by parameter descriptors in the ENTRY attribute of the variable's declaration. An internal entry (and its parameters) must not be declared explicitly in the containing procedure.
- Each parameter must have a corresponding argument, at the time of the procedure's invocation. PL/I matches the data type of the parameter with the data type of the corresponding argument and creates a dummy argument if they do not match. (See Section 13.1.8.2.)

Array Parameters

If the name of an array variable is passed as an argument, the corresponding parameter descriptor or parameter declaration must specify the same number of dimensions as the argument variable. You can declare the bounds of a dimension for an array parameter using asterisks (*) or optionally signed integer constants. If the bounds are specified with integer constants, they must match exactly the bounds of the corresponding argument. An asterisk indicates that the bounds of a dimension are not known. (If one dimension contains an asterisk, all the dimensions must contain asterisks.) For example:

```
DECLARE SUMUP ENTRY ((*) FIXED BINARY);
```

This declaration indicates that SUMUP's argument is a one-dimensional array of fixed-point binary integers that can have any number of elements. Any one-dimensional array of fixed-point binary integers may be passed to this procedure.

All the data type attributes of the array argument and parameter must match.

Arrays are always passed by reference. They cannot be passed by dummy argument.

Structure Parameters

If the name of a structure variable is passed as an argument, the corresponding parameter descriptor or declaration must be identical, in terms of structure levels, members' sizes, and members' data types. The level numbers do not have to be identical but the levels must be logically equivalent. You can specify array bounds and string lengths with asterisks or with

optionally signed integer constants. The following example shows the parameter descriptor for a structure variable:

```
DECLARE SEND_REC ENTRY (1,
                          2 FIXED BINARY(31),
                          2 CHARACTER(40) VARYING,
                          2 PICTURE '999V99');
```

The written argument in the invocation of the external procedure SEND_REC must have the same structure, and its members must have the same data types.

Structures are always passed by reference. They cannot be passed by dummy argument.

Character-String Parameters

If a character-string variable is passed as an argument, the corresponding parameter descriptor or parameter declaration can specify the length using an asterisk (*) or an optionally signed nonnegative integer constant. For example:

```
COPYSTRING: PROCEDURE (INSTRING,COUNT);
DECLARE INSTRING CHARACTER(*);
```

The asterisk in the declaration of this parameter indicates that the string may have any length. The string is fixed length unless VARYING is also included in the declaration.

Entry, File, and Label Constant Parameters

Entry, file, and label constants may be passed as arguments. The actual parameter is a variable.

13.1.8.2 Argument Passing

The following paragraphs describe the precise rules that determine how PL/I passes an argument to procedures written in PL/I. For rules and details on passing arguments to procedures written in other languages, see Section 13.2.2.

Number of Arguments

The number of arguments in the argument list must equal the number of parameters of the invoked entry point. The compiler checks that the count matches as follows:

- For an internal procedure, the compiler checks the number of arguments in the argument list against the number of parameters on the PROCEDURE or ENTRY statement for the internal procedure.
- For an external procedure, the compiler checks that the number of parameter descriptors in the ENTRY declaration list matches the number of arguments in the procedure invocation.

Actual Arguments

When a PL/I procedure is invoked, each of its parameters is associated with a variable determined by the corresponding written argument of the procedure call. This is the actual argument for this procedure invocation. It may be

- A reference to the written argument.
- A dummy argument.

The data type of the actual argument is the same as that of the corresponding parameter. When a written argument is a variable reference, PL/I matches the variable against the corresponding parameter's data type according to the rules given under the heading "Argument Matching," below. If they match, the actual argument is the variable denoted by the written argument. That is, the parameter denotes the same storage as the written variable reference. If they do not match, the compiler creates a dummy argument and assigns to it the value of the written argument.

Dummy Arguments

A dummy argument is a unique variable allocated by the compiler, which exists only for the duration of the procedure invocation.

When the written argument is a constant or an expression, the actual argument is always a dummy argument. The value of the written argument is assigned to the dummy argument before the call. The data type of the written argument must be valid for assignment to the data type of the dummy argument.

Aggregate Arguments

An array, structure, or area argument must be a variable reference that matches the corresponding parameter. It may not be a reference to an unconnected array. A dummy argument is never created for an array, structure, or area.

Argument Matching

A written argument that is a variable reference is passed by reference only if the argument and the corresponding parameter have identical data types:

- For an internal procedure, the attributes of the argument must match those specified in the declaration of the parameter.
- For an external procedure or a procedure invoked via an ENTRY variable, the attributes specified in the ENTRY attribute parameter descriptor must match those of the arguments.

When the compiler detects that a scalar variable argument does not match the data type of the corresponding parameter, it issues a warning message, creates a dummy argument, and associates the address of the dummy argument with the corresponding parameter. You can suppress the warning message and force the creation of a dummy argument if you enclose the argument in parentheses. For example, if a parameter requires a CHARACTER VARYING string and an argument is a CHARACTER non-varying variable, you would enclose the variable in parentheses.

For string lengths and array bounds, an asterisk (*) in the parameter matches any expression. An integer constant matches only an integer constant with the same value.

Conversion of Arguments

When the data type of a written argument is suitable for conversion to the data type of the corresponding parameter descriptor, PL/I performs the conversion of the argument to a dummy argument using the rules described in Section 12.4.3 and Appendix A.

13.2 Calling External Procedures

An external procedure is one whose text is not contained in any other block. The source text of an external procedure can be compiled separately from that of a calling procedure. The primary coding differences between internal and external procedures are

- Before an external procedure can be invoked (except via an entry variable), its name must be declared within the procedure that invokes it. The DECLARE statement for the external entry name must also provide a list of parameter descriptors that give the data type(s) of the parameters that the procedure requires, if any, as well as a RETURNS attribute for a function procedure.

  Internal procedures must not be explicitly declared. The procedure name is implicitly declared by its occurrence in the PROCEDURE or ENTRY statement.

- External procedures can reference the same variable only if it is declared with the EXTERNAL attribute in all of them.

  An internal procedure, on the other hand, can reference internal variables declared in any procedure in which it is contained.

- Any procedure can call an external procedure.

  An internal procedure can be called only by the procedure that contains it or by other procedures at the same level of nesting within the containing procedure. The only exception is invocation via an entry variable.

The following example illustrates the use of an external procedure:

```
WINDUP: PROCEDURE;
    .
    .
    .
DECLARE PITCH EXTERNAL ENTRY (CHARACTER(15) VARYING,
                               FIXED BINARY(7) );
    .
    .
    .
CALL PITCH (PLAYER_NAME,NUMBER_OF_OUTS);
```

The procedure WINDUP declares the procedure PITCH with the EXTERNAL and ENTRY attributes. The text of PITCH is in another source program that is separately compiled. When the object module that contains WINDUP is linked, the linker must be able to locate the object module that contains PITCH. This can be accomplished by including both object modules in the LINK command line, or by placing PITCH in an object module library and including the library in the LINK command line.

When a CALL statement or function reference invokes an entry point in an external procedure, the entry constant must be declared with the ENTRY attribute, as in the example above. Such a declaration must also describe the parameters for that entry point, if any. For example:

```
DECLARE PITCH ENTRY (CHARACTER(*), FIXED BINARY(15));
```

The identifier PITCH is declared as an entry constant (in this context, ENTRY external). When the procedure containing this declaration is linked to other procedures, one of them must define an entry point named PITCH as the label either of a PROCEDURE statement or an ENTRY statement. If the linker cannot locate an external entry point, it issues a warning message.

The parameter descriptors define the data types of the parameters for the entry point PITCH. Arguments of these types must be supplied when PITCH is invoked.

If PITCH is to invoke a function, the DECLARE statement must also include a RETURNS attribute describing the attributes of the returned value, such as

```
DECLARE PITCH ENTRY (CHARACTER(*), FIXED BINARY(15))
    RETURNS(FIXED);
```

Within the scope of this DECLARE statement, the entry constant PITCH must be used in a function reference. The function reference will invoke the external entry point, and a returned fixed-point binary value will become the value of the function reference.

Section 13.2.1 provides a full description of entry data and the declaration of entry constants.

A PL/I program can invoke an external procedure that is not written in PL/I. A common instance is the use of a VAX/VMS system service by a PL/I program to obtain some system function not available directly through PL/I. Or, a PL/I program can invoke an external procedure written in another language that provides an application-specific function. Such instances are possible because of the VAX-11 Calling Standard, a set of conventions for passing arguments among procedures. Section 13.2.2 describes these conventions.

13.2.1 Entry Data

Entry constants and variables invoke procedures through specified entry points. An entry value specifies an entry point and a block activation of a procedure.

No conversions are defined between entry data and other data types. An entry variable can be assigned only the value of an entry constant or the value of another entry variable. The only valid operations for entry data are comparisons for equality (=) and inequality (^=); two entry values are equal if they refer to the same entry point in the same block activation.

13.2.1.1 Entry Constants

You declare entry constants implicitly when you write labels on PROCEDURE or ENTRY statements.

Internal entry constants are declared by writing labels on PROCEDURE or ENTRY statements whose procedure blocks are nested in another block. An internal entry constant can be used anywhere within the containing block to invoke its procedure block. You cannot explicitly declare an internal entry constant in the containing block.

External entry constants are declared by writing labels on PROCEDURE or ENTRY statements that belong to external procedures, and by explicitly declaring the name with the ENTRY attribute in the calling procedure. You can use an external entry constant to invoke its procedure block from any program location within its scope, which is either the scope of its declaration (as a label in the external procedure) or the scope of a DECLARE statement for the constant (in the calling procedure).

The declaration of an external entry constant gives the compiler the information it needs to invoke a separately compiled procedure. The declaration must agree with the actual entry point: it must contain parameter descriptors for any parameters specified at the entry point; and, if the entry constant is to be used in a function reference, the declaration must have a

returns descriptor describing the returned value. The format for a declaration of an external entry constant is

```
DECLARE identifier ENTRY [ (parameter-descriptor,...) ]

                [ OPTIONS (VARIABLE)            ]
                [ RETURNS (returns-descriptor)  ] ;
```

identifier

The label associated with the external entry point.

parameter-descriptor

A set of attributes describing a parameter of the specified entry. The attributes of a single parameter must be separated by spaces; sets of attributes (each set describing a different parameter) must be separated by commas.

OPTIONS (VARIABLE)

An option indicating that the specified external procedure can be invoked with a variable number of arguments. This option is provided for use in calling non-PL/I procedures.

RETURNS (returns-descriptor)

An option giving, for an entry invoked as a function reference, the data type attributes of the function value returned. For such entry points, the RETURNS attribute is required; for those invoked by CALL statements, the RETURNS attribute is invalid. Section 13.1.7 describes the syntax of this option.

The following example declares the external entry constant COPYSTRING:

```
DECLARE COPYSTRING ENTRY (CHARACTER (40) VARYING,
                          FIXED BINARY(7))
        RETURNS (CHARACTER(*));
```

This entry has two parameters: (1) a varying-length character string with a maximum length of 40 and (2) a fixed-point binary value. The RETURNS attribute indicates that COPYSTRING is invoked as a function and that it returns a character string with any length. COPYSTRING might look like this:

```
COPYSTRING: PROCEDURE (INSTRING, ITERATIONS)
        RETURNS (CHARACTER (*));
DECLARE INSTRING CHARACTER (40) VARYING,
        ITERATIONS FIXED BINARY (7),
        OUTSTRING CHARACTER (40);
            .
            .
            .
        RETURN (OUTSTRING);
        END;
```

13.2.1.2 Entry Variables

Entry variables are those (including parameters) that take entry values. If you specify the VARIABLE attribute with the ENTRY attribute in a DECLARE statement, or if the declared identifier occurs in a parameter list, the declared identifier is an entry variable. You can assign an entry constant to an entry variable, or you can assign to it the value of another entry variable.

When you use an entry variable to invoke a procedure, its declaration must agree with the definition of the entry point: the parameter descriptor for the entry variable must match the parameter descriptor on the declaration of the entry constant.

The scope of an entry variable name can be either INTERNAL or EXTERNAL. If you specify neither with ENTRY VARIABLE, the default is INTERNAL.

You can use an entry variable to represent different entry points during the execution of the PL/I program. For example:

```
DECLARE E ENTRY (FIXED BINARY (7)) VARIABLE,
        (A,B) ENTRY (FIXED BINARY (7));

        E = A;
        CALL E (10);
```

In this example, the entry constant A is assigned to the entry variable E. The CALL statement results in the invocation of the external entry point A.

13.2.2 Passing Arguments to Non-PL/I Procedures

There are three ways that a PL/I procedure can pass an argument to a non-PL/I procedure. They are

- By immediate value. The actual value of the argument is passed.
- By reference. The address in storage of the argument is passed.
- By descriptor. The address in storage of a data structure describing the argument is passed.

The following sections describe the requirements for each of these argument-passing mechanisms.

13.2.2.1 Passing Arguments by Immediate Value

You must use the VALUE attribute in a parameter descriptor for an argument to be passed by immediate value. The following declaration of the external entry SYS$SETEF illustrates such a descriptor:

```
DECLARE SYS$SETEF ENTRY (FIXED BINARY(31) VALUE);
```

This declaration of the Set Event Flag system service (SYS$SETEF) specifies a single argument to be passed by immediate value.

Arguments that can be passed by immediate value are limited to the following data types, which can be expressed in 32 bits:

- FIXED BINARY (31)
- BIT (32) ALIGNED
- ENTRY
- OFFSET
- POINTER

When you specify the VALUE attribute in a parameter descriptor, you can specify the ANY attribute instead of declaring any data type attributes. For example, the declaration of SYS$SETEF can appear as follows:

```
DECLARE SYS$SETEF ENTRY (ANY VALUE);
```

At the time of the procedure's invocation, PL/I converts the written argument as needed to create a longword dummy argument.

13.2.2.2 Passing Arguments by Reference

By default, PL/I passes all arguments except character strings and arrays with nonconstant extents by reference. The parameter descriptor for an argument to be passed by reference need specify only the data type of the parameter.

For example, the Read Event Flags (SYS$READEF) system service requires that its first argument be passed by immediate value and its second, by reference. You could declare this procedure as follows:

```
DECLARE SYS$READEF ENTRY (FIXED BINARY(31) VALUE,
                          BIT (32) ALIGNED);
```

When the procedure is invoked, the second argument must be a variable declared as BIT(32) ALIGNED. PL/I passes the argument by reference.

An argument of any data type may be passed by reference. Bit-string variables, however, must have the ALIGNED attribute.

You must always ensure that the data types in the parameter descriptors of all output arguments match the data types of the written arguments. For convenience, you can specify ANY in the parameter descriptor. To describe an argument to be passed by reference, you can specify the ANY attribute without the VALUE attribute. The argument can be of any addressable data type known to PL/I. For example, you could declare the SYS$READEF service as follows:

```
DECLARE SYS$READEF ENTRY (FIXED BINARY(31) VALUE, ANY);
```

The second parameter descriptor in the ENTRY attribute indicates that the second argument is to be passed by reference to the procedure SYS$READEF, and that it can have any data type. When you specify

ANY for an argument to be passed by reference, you cannot specify data type attributes. Note that if you specify the VALUE attribute in conjunction with the ANY attribute, PL/I passes the argument by immediate value.

The ANY attribute is especially useful when you must specify a data structure as an argument. You need not declare the structure within the parameter descriptor, only the ANY attribute.

When an argument is passed by reference, PL/I passes the address of the actual argument. This address can be interpreted as a pointer value. In fact, you can explicitly specify a pointer value as an argument for data to be passed by reference. For example:

```
DECLARE SYS$READEF (ANY VALUE, POINTER VALUE),
        FLAGS BIT(32) ALIGNED;

CALL SYS$READEF (4, ADDR(FLAGS));
```

At this procedure invocation, PL/I places the pointer value returned by the ADDR built-in function directly in the argument list.

13.2.2.3 Passing Arguments by Descriptor

A descriptor is a structure that describes the data type, extents, and address of a data item. When passing an argument by descriptor, PL/I creates the descriptor and places its address in the argument list for the called procedure.

PL/I passes arguments by descriptor when a parameter descriptor specifies the following:

- A character string with an asterisk length or an array with asterisk extents
- An unaligned bit string or an array or structure consisting entirely of unaligned bit strings
- A structure containing any strings or arrays with asterisk extents
- ANY without VALUE, and the corresponding written argument is specified using the DESCRIPTOR built-in function

For example, PL/I passes by descriptor the arguments associated with the following parameter descriptors:

```
DECLARE UNSTRING ENTRY (CHARACTER(*)),
        TESTBITS ENTRY (BIT(3)),
        MODEST ENTRY (1,
                        2 CHARACTER(*),
                        2,
                          3 BIT(3),
                          3 BIT(3));
```

When you declare a non-PL/I procedure that requires a character-string descriptor for an argument, specify the parameter descriptor as CHARACTER(*). For example, the Set Process Name (SYS$SETPRN) system service requires the address of a character-string descriptor as an argument. You can declare this service as follows:

```
DECLARE SYS$SETPRN ENTRY (CHARACTER(*));
```

When a parameter is declared as CHARACTER(*), its written argument can be

- A character-string constant or expression.
- A fixed-length character-string variable.
- A varying character-string variable or a variable declared as CHARACTER(*)VARYING.

For any of those arguments, PL/I constructs a character-string descriptor and passes its address.

To force an argument to be passed by descriptor, use the DESCRIPTOR built-in function. For example:

```
DECLARE P ENTRY (ANY);
DECLARE (X,Y) FIXED DECIMAL (7,2);

CALL P(DESCRIPTOR (X));
CALL P(Y);
```

Here, X is passed by descriptor as specified by the DESCRIPTOR built-in function. Y is passed by reference. Section 19.2 contains the syntax and rules for the DESCRIPTOR built-in function.

Chapter 14

Program Control

The statements described in this chapter allow your program to repeat sequences of operations, to transfer control or select operations based on the result of a test, and to terminate. They are the DO, BEGIN, END, IF, SELECT, GOTO, LEAVE, STOP, and null statements.

14.1 DO Statement

The DO statement defines the beginning of a sequence of statements to be executed in a group. The group ends with the nonexecutable statement END. DO-groups have several formats, which are described individually under the following subheadings:

- Simple DO
- DO WHILE
- DO UNTIL
- Controlled DO
- DO REPEAT

14.1.1 Simple DO

A simple DO statement is noniterative. Its format is

```
DO;
    .
    .
    .
END;
```

The statements that appear between the DO statement and its corresponding END statement are executed once, after which control passes to the next executable statement in the program.

For example:

```
IF A < B THEN DO;
   PUT LIST ('More data needed');
   GET LIST (VALUE);
   A = A + VALUE;
   END;
```

The most common use of the simple DO statement is as the action of the THEN clause of an IF statement, as shown above, or of an ELSE option.

14.1.2 DO WHILE

A DO WHILE statement executes a group of statements as long as a particular condition is satisfied. When the condition is not true, the group is not executed and control passes to the next executable statement in the program. This format of the DO statement is

```
DO WHILE (test-expression);
   .
   .
   .
END;
```

test-expression

Any expression that yields a scalar bit-string value. If any bit of the value is a 1, then the test expression is true; otherwise, the test expression is false. The test expression must be enclosed in parentheses.

This expression is evaluated before each execution of the DO-group. It must have a true value in order for the DO-group to be executed. Otherwise, control passes outside of the DO-group to the next executable statement after the END statement that terminates the group.

The following examples illustrate the use of the DO WHILE statement.

```
DO WHILE (A < B);
```

This DO-group executes as long as the value of the variable A is less than the value of the variable B.

```
DO WHILE (LIST->NEXT ^= NULL());
```

This DO-group executes while a forward pointer in a linked list has a value.

```
DECLARE EOF BIT(1) INITIAL('0'B);
    .
    .
    .
ON ENDFILE(INFILE) EOF = '1'B;

DO WHILE (^EOF);
        READ FILE(INFILE) INTO(INREC);
    .
    .
    .
END;
```

This DO-group reads records from the file INFILE until the end of the file is reached. At the beginning of each iteration of the DO-group, the expression ^EOF is evaluated; the expression is true until the ENDFILE ON-unit sets the value of EOF to '1'B.

14.1.3 DO UNTIL

A DO UNTIL statement executes a group of statements until a particular condition is satisfied. That is, while the condition is false, the group is repeated. The format of the DO UNTIL statement is

```
DO UNTIL (test-expression);
    .
    .
    .
END;
```

test-expression

Any expression that yields a scalar bit-string value. If any bit of the value is 1, then the test expression is true; otherwise the test expression is false. A false value is necessary for the DO-group to be repeated. Otherwise, control passes to the next executable statement following the END statement that terminates the DO-group. The test expression must be enclosed in parentheses.

The following examples illustrate the use of the DO UNTIL statement.

```
DO UNTIL (K<ALPHA);
```

This DO-group is executed at least once and then repeats as long as the value of the variable K is greater than or equal to the value of the variable ALPHA.

```
DO UNTIL (LIST ->NEXT = NULL())
```

This DO-group is executed until a forward pointer in a linked list has a null value.

```
DECLARE STR BIT (8) CONTROLLED;
    .
    .
    .
ALLOCATE  STR;  /* 1st allocation */
    .
    .
    .
ALLOCATE STR;   /* nth allocation */
    .
    .
    .
DO UNTIL (ALLOCATION(STR)=0);
   PUT SKIP LIST (STR);
   FREE STR;
   END;

END;
```

This DO-group frees bit strings from storage until all generations have been released. Because the UNTIL option is always executed at least once, at least one generation must be allocated; otherwise the ERROR condition is raised. At the end of each repetition of the DO-group, the status of the generations is checked with the ALLOCATION built-in function. A null string terminates the execution of the group and passes control to the next executable statement after the first END statement.

14.1.4 Controlled DO

A controlled DO statement identifies a variable whose value controls the execution of the DO-group, and defines the conditions under which the control variable is to be modified and tested. When the value of the control variable exceeds the specified end value, control passes out of the DO-group. A WHILE or UNTIL clause may also be included. The WHILE expression is evaluated before each iteration, including the first, but after assignment to the control variable. The UNTIL expression is evaluated after each iteration, including the first, but before assignment to the control variable. The format of the controlled DO statement is

```
DO control-variable = start-value
{ TO end-value [BY modify-value] }
{ BY modify-value                }
  [WHILE (test-expression) ]
  [UNTIL (test-expression) ];
  .
  .
  .
END;
```

A controlled DO statement that does not specify a TO or BY option results in a single iteration of the following DO-group. Since there is no TO or BY expression to change the value of the variable, the DO-group will not be executed again.

control-variable

A reference to a variable whose current value determines whether the DO-group is executed. The control variable must be of an arithmetic data type.

start-value

An expression specifying the initial value to be given to the control variable. Evaluation of this expression must yield an arithmetic value.

end-value

An expression giving the value to be compared with the control variable during successive iterations. Evaluation of this expression must yield an arithmetic value.

modify-value

An expression giving a value by which the control value is to be modified. Evaluation of this expression takes place once when control first reaches the DO statement, and must yield an arithmetic value. If the BY option is not specified, the modify value is 1 by default.

WHILE(test-expression)
UNTIL(test-expression)

Options specifying conditions that further control the execution of the DO-group. See the preceding sections for details.

The following examples illustrate the controlled DO statement.

```
DO I = 2 TO 100 BY 2;
```

This DO-group executes 50 times, with values for I of 2, 4, 6, and so on.

```
DO I = LBOUND(ARRAY,1) TO HBOUND(ARRAY,1);
```

This DO-group executes as many times as there are elements in the array variable ARRAY, using the subscript values of the array's elements.

```
DO I = 1 BY 1 WHILE (X < Y);
```

This DO-group continues executing with successively higher values for I until the value of the variable X equals or is greater than the value of the variable Y.

```
DO I = 1 BY -1 UNTIL (X < Y);
```

This DO-group continues executing with successively lower values for I while the value of the variable X is equal to or greater than the value of the variable Y.

14.1.5 DO REPEAT

The DO REPEAT statement executes a DO-group repetitively for different values of a variable. The variable is assigned a start value that is used on the first iteration of the group. The REPEAT expression is evaluated before each subsequent iteration, and its result is assigned to the variable. A WHILE clause may also be included; if it is, the WHILE expression is evaluated before each iteration, including the first but after assignment to the variable. The format of the DO REPEAT statement is

```
DO variable = start-value REPEAT (expression)
   [WHILE (test-expression) ]
   [UNTIL (test-expression) ];
      .
      .
      .
END;
```

variable

A reference to a scalar variable of any type.

start-value

An expression specifying the initial value to be given to the variable. The evaluation of this expression must yield a value that is valid for assignment to the variable.

expression

An expression giving the value to be assigned to the variable on reiterations of the DO REPEAT group. The expression is evaluated before each reiteration. Evaluation of this expression must yield a result that is valid for assignment to the variable.

WHILE(test-expression)
UNTIL(test-expression)

Options specifying conditions that control the termination of the DO REPEAT group. The specified test expression must yield a scalar bit-string value. If any bit of the value is 1, then the test expression is true; otherwise, the test expression is false. The test expression must be enclosed in parentheses.

The WHILE test expression is evaluated each time control reaches the DO statement; a true value is necessary for the DO-group to be executed. Otherwise, control passes outside of the DO-group to the next executable statement following the END statement that terminates the group.

Both the WHILE and UNTIL options check the status of test expressions, but they differ in that the WHILE option tests the value of the test expression at the beginning of the DO-group, and the UNTIL

option tests the value of the test expression at the end of the DO-group. Therefore, a DO-group with the UNTIL option will always be executed at least once, but a DO-group with the WHILE option may never be executed.

NOTE

> If the WHILE or UNTIL options are omitted, the DO REPEAT statement specifies no means for terminating the group; the execution of the group must be terminated by a statement or condition occurring within the group, such as a GOTO statement, LEAVE statement, or an ENDFILE condition.

The following examples illustrate the use of the DO REPEAT statement:

```
DO LETTER='A' REPEAT (BYTE(I));
```

Here, the group will be repeated with an initial LETTER value of 'A' and with subsequent values assigned by the built-in function BYTE(I). The variable I may be assigned new values within the group. The group will iterate endlessly unless terminated by a statement or condition within the group.

```
DO I = 1 REPEAT (I + 2) WHILE ( I <= 100 );
```

This example has the same effect as the controlled DO statement:

```
DO I = 1 TO 100 BY 2;
```

The most common use of the DO REPEAT statement is in the manipulation of lists. For example:

```
DO P = LIST_HEAD REPEAT (P->LIST.NEXT)
    WHILE ( P ^= NULL() );
```

In this example, the pointer P is initialized with the value of the pointer variable LIST_HEAD. The DO-group is executed with this value of P. The REPEAT option specifies that, each time control reaches the DO statement after the first execution of the DO-group, P is to be set to the value of LIST.NEXT in the structure currently pointed to by P.

Both WHILE and UNTIL may be used in combination to check the status of a DO-group both before and after execution.

14.2 BEGIN Statement

The BEGIN statement denotes the start of a begin block. Its format is

```
BEGIN;
```

A begin block is a sequence of statements headed with a BEGIN statement and terminated by an END statement. In general, a begin block can be used wherever a single executable statement is valid, for instance, in an

ON-unit. The statements in a begin block can be any PL/I statements, and begin blocks can contain DO-groups, DECLARE statements, procedures, and other (nested) begin blocks.

A begin block provides a convenient way to localize variables. Those declared as internal variables within a begin block are not allocated storage until the block is activated. When the block terminates, storage for internal automatic variables is released. A begin block terminates when

- Its corresponding END statement is encountered. Control continues with the next executable statement in the program.
- It executes a nonlocal GOTO to transfer control to a previous block.
- It executes a RETURN statement.

A begin block differs from a DO-group chiefly in its ability to localize variables. Variables declared within DO-groups are not localized to the group (unless, of course, the group contains a begin block or procedure that declares internal variables). Begin blocks are preferable when you want to restrict the scope of variables; furthermore, there are some cases (such as ON-units) in which DO-groups cannot be used. Otherwise, DO-groups are often more efficient, because they do not have the overhead associated with block activation. In general, you should use a DO-group instead of a begin block unless there are declarations present or you require multiple statements in an ON-unit.

A begin block can designate a series of statements to be executed depending on the success or failure of a test in an IF statement. For example:

```
IF A = B THEN BEGIN ;
   .
   .
   .
   END;
```

A begin block also provides the only way to denote a series of statements to be executed when an ON condition is signaled. For example:

```
ON ERROR BEGIN;
    [statement ...]
    END;
```

14.3 END Statement

The END statement marks the end of the block or group headed by the most recent BEGIN, DO, SELECT, or PROCEDURE statement. Its format is

```
END [identifier];
```

identifier

An optional reference to the unsubscripted label on the PROCEDURE, BEGIN, SELECT, or DO statement terminated by the END statement. If specified, the identifier must match the label of the most recent BEGIN, DO, SELECT, or PROCEDURE statement that is not already matched with an END statement. If the identifier is omitted, the most recent statement is matched by default. The identifier cannot be a reference to a subscripted label.

Note that a procedure invoked as a function must execute a RETURN statement before it encounters the END statement marking the end of the procedure.

When the END statement is encountered, one of the following actions is performed, depending on the type of block or group that it terminates:

- When an END statement denotes the end of a procedure, it is terminated. The storage allocated for the block is released, and all automatic variables are made inaccessible. If the current procedure is the main or only procedure, the program terminates. Otherwise, control returns to the statement following the CALL statement that invoked the procedure.
- When an END statement denotes the end of a begin block, it is terminated. Storage allocated for the block is released, and all automatic variables are made inaccessible. Control passes to the next executable statement.
- When an END statement denotes the end of a DO-group, control returns either to the DO statement that heads the group or to the next executable statement following the END statement. If the DO-group is headed by a simple DO, that is, one that causes the DO-group to be executed only once, control passes to the next executable statement. Otherwise, control returns to the head of the DO-group, where the control variable or expression is tested.

14.4 IF Statement

The IF statement tests an expression and performs a specified action if the result of the test is true. Its format is

```
IF test-expression THEN action [ELSE action];
```

test-expression

Any valid expression that yields a scalar bit-string value. If any bit of the value is 1, then the test expression is true; otherwise, the test expression is false.

action

Any of the following:

- Any unlabeled statement except a DECLARE, FORMAT, PROCEDURE, END, or ENTRY statement
- An unlabeled DO-group, SELECT-group, or begin block

The IF statement evaluates the test expression. If the expression is true, the action specified following the keyword THEN is executed. Otherwise, the action, if any, specified following the ELSE keyword is executed. For details on the syntax of specifying expressions, see Chapter 12.

The following examples illustrate the use of the IF statement.

```
IF A < B THEN BEGIN;
```

The begin block after this statement is executed if the value of the variable A is less than the value of the variable B.

```
IF ^SUCCESS THEN
         CALL PRINT_ERROR;
      ELSE
         CALL PRINT_SUCCESS;
```

The IF statement defines action to be taken if the variable SUCCESS has a false value (the THEN clause) and an action to be taken otherwise (the ELSE clause).

You can nest IF statements; that is, the action specified in a THEN or an ELSE clause may be another IF statement. An ELSE clause is matched with the nearest preceding IF/THEN that is not itself matched with a preceding ELSE. For example:

```
IF ABC
   THEN IF XYZ
           THEN GOTO GBH;
      ELSE GOTO THESTORE;
   ELSE GOTO HOME;
```

The first ELSE clause is executed if ABC is true and XYZ is false. The second ELSE clause is executed if ABC is false.

In some cases, proper matching of IF and ELSE may require a null statement as the target of an ELSE. For example:

```
IF ABC
   THEN IF XYZ THEN GOTO HOME;
         ELSE;
   ELSE GOTO THESTORE;
```

The ELSE GOTO THESTORE statement is executed if ABC is false.

14.5 SELECT Statement

The SELECT statement tests a series of expressions and performs specified action if the result of any of the tests is true. The format for the SELECT statement is

```
SELECT;
    WHEN (condition-list) select-action;
    WHEN (condition-list) select-action;
        .
        .
        .
    WHEN (condition-list) select-action;
    [OTHERWISE select-action];
END;
```

condition-list

One or more test expressions that yield a scalar bit-string value. If any bit of the value is 1, then the test expression is true; otherwise the test expression is false.

The format of a condition list is

```
(expression[,...])
```

select-action

Any of the following:

- Any statement except an END, ENTRY, DECLARE, FORMAT, or PROCEDURE statement.
- A DO-group, SELECT-group, or a BEGIN-END block.

If any of the test expressions in the condition list results in a bit string containing any bit with the value of '1'B, the action specified following the keyword WHEN is executed. No further expressions in that WHEN clause or subsequent WHEN clauses are evaluated, and no subsequent select actions are executed.

If none of the expressions in the WHEN clauses evaluates to a bit string containing '1'B, the action specified after the OTHERWISE clause is executed unconditionally. If the OTHERWISE clause is omitted and the evaluation of the SELECT-group does not result in the selection of any action, an ERROR condition is raised.

After execution of a select action following a WHEN or OTHERWISE clause, control passes to the next executable statement following the END statement that terminates the SELECT-group, unless normal flow is altered within the select action.

For example:

```
SELECT;
   WHEN (A=50) B=B+1;
   WHEN (A=60) B=B+2;
   WHEN (A=70) B=B+3;
   WHEN (A=80) B=B+4;
   WHEN (A=90) B=B+5;
   OTHERWISE B=B+C;
END;
```

The SELECT statement defines action to be taken if the variable A has any of the values specified in the WHEN clauses. If none of the WHEN clauses are true, the action specified in the OTHERWISE clause is executed.

The action specified in a WHEN or OTHERWISE clause may be another SELECT statement, resulting in nested SELECT statements. Then, the OTHERWISE clause is matched to the nearest preceding SELECT/WHEN. For example:

```
SELECT;
   WHEN (condition A)
        SELECT;
           WHEN (condition-A1) statement 1;
           WHEN (condition A2) statement 2;
           END;
   WHEN (condition B)
        SELECT;
           WHEN (condition B1) statement 3;
           WHEN (condition B2) statement 4;
           OTHERWISE statement 5;
           END;
   OTHERWISE statement 6;
END;
```

In this example, the first statement is executed when both A and A1 conditions are true. The second statement is executed when both A and A2 conditions are true. If none of the A conditions are satisfied, B conditions are checked. Statement 5 is executed when condition B is true, but conditions B1 and B2 are false. Statement 6 is executed when all of the preceding conditions are false.

Conditions are checked from top to bottom; thus, if two conditions are true, only the first course of action is taken.

Notice that an END statement follows statements 2 and 5: you must close each group in nested SELECT-groups. Also notice that there is no OTHERWISE after statement 2: you do not have to include an OTHERWISE clause, but an ERROR could result if none of the WHEN clauses are selected.

In some cases, proper matching of WHEN and OTHERWISE may require a null statement as the target of OTHERWISE. For example:

```
SELECT;
    WHEN (condition A,B,C) GOTO FILE_READ;
    WHEN (condition D,E) GOTO UPDATE;
    OTHERWISE;
END;
```

In this example, control is passed to the next executable statement after END if conditions A, B, C, D, and E are not true.

14.6 GOTO Statement

The GOTO statement causes control to be transferred to a labeled statement in the current or any outer procedure. Its format is

```
{ GOTO  }  label-reference;
{ GO TO }
```

label-reference

A label constant or an expression that, when evaluated, yields a label value. A label value denotes a statement in the program and a block activation. The specified label cannot be the label of an ENTRY, FORMAT, or PROCEDURE statement.

If the specified label value is not in the current block, the GOTO statement is considered nonlocal. The following can occur:

- The current block, and any blocks intervening between it and the block containing the label value, are released. This rule applies to procedure blocks and to begin blocks.
- If a GOTO statement transfers control out of a procedure that is invoked in a function reference, the statement containing the function reference is not evaluated further.

A label consists of any valid identifier terminated by a colon. A name occurring as a statement label is implicitly declared as a label constant, with the attributes LABEL and constant. Label constants may not be explicitly declared.

The following restrictions apply to the use of labels and label data:

- No statement can have more than one label. However, an executable statement can be preceded by any number of labeled null statements, which have the same effect as would multiple labels.
- Operations on label values are restricted to the operators = and ^=, for testing equality or inequality. Two values are equal if they refer to the same statement in the same block activation.

- Any statement in a PL/I program can be labeled except
  - A DECLARE statement.
  - A statement beginning an ON-unit, THEN clause, ELSE clause, WHEN clause, or OTHERWISE clause.
- Labels on PROCEDURE, ENTRY, and FORMAT statements are not considered statement labels and may not be used as the targets of GOTO statements.
- An identifier occurring as a label in a block may not be declared in that block (except as a structure member) or occur in the block's parameter list.
- Any reference to a label value after its block activation terminates is an error with unpredictable results.

This example demonstrates the use of the GOTO statement:

```
ON ERROR GOTO ERROR_MESSAGE;
```

The GOTO statement provides a transfer address for the current procedure when the ERROR condition is signaled.

The following subsections describe label array constants, which allow you to write labels with constant subscripts, and label variables, which can be assigned values and then used in GOTO statements to provide flexibility.

14.6.1 Label Array Constants

Any label constant except the label of a PROCEDURE or FORMAT statement can have a single subscript. Subscripts must be specified using integer constants or constant identifiers, which must appear in parentheses following the label name. For example:

```
PART(1):
    .
    .
    .
PART(2):
    .
    .
    .
```

When labels are written as shown here, the unsubscripted label name represents the implicit declaration of a label array constant. In this example, the array is named PART and is treated as if it were declared within the block containing the subscripted labels. Elements of this array may be referenced in GOTO statements that specify a subscript, for example:

```
GOTO PART(I);
```

where I is a variable whose value represents the subscript of the element of PART that is the label to be given control.

14.6.2 Label Variables

When an identifier is explicitly declared with the LABEL attribute, it acquires the VARIABLE attribute by default. Such a variable can be used to denote different label values during the execution of the program. The following examples demonstrate the use of label variables.

```
DECLARE PROCESS LABEL;
   .
   .
   .
   IF CODE THEN
      PROCESS = BILLING;
   ELSE
      PROCESS = CHARGE;
   .
   .
   .
GOTO PROCESS;
```

When the GOTO statement evaluates the reference to the label PROCESS, the result is the current value of the variable. The GOTO statement transfers control to either of the labels BILLING or CHARGE, depending on the current value of the Boolean variable CODE.

```
%REPLACE REMOVE_TEXT BY 2;
   .
   .
   .
DECLARE PROCESS(5) LABEL VARIABLE;
   .
   .
   .
GOTO PROCESS(REMOVE_TEXT);
```

The GOTO statement evaluates the label reference and transfers control to the label constant corresponding to the second element of the array PROCESS. PROCESS consists of label variables.

14.7 LEAVE Statement

The LEAVE statement causes control to be transferred out of the immediately containing DO-group or out of the containing DO-group whose label is specified with the statement. The format of the LEAVE statement is

LEAVE [label-reference];

label-reference

A reference to a label on a DO statement that heads a containing DO-group. The label reference can be a label constant or a subscripted label constant for which the subscript is specified with an integer

constant. The label reference cannot be a label variable, nor can it be a subscripted label constant for which the subscript is specified with a variable.

Upon execution, a LEAVE statement with no label reference causes control to be transferred to the first statement following the END statement that terminatesthe immediately containing DO-group. If the LEAVE statement has a label, control is passed to the first executable statement following the end statement for the corresponding label indicated in the LEAVE statement. Thus, the LEAVE statement provides an alternative means of terminating execution of a DO-group. In the case of a LEAVE statement with a label reference, several nested DO-groups can be terminated as control is transfered outside the referenced DO-group.

The following restrictions apply to the use of the LEAVE statement:

- A LEAVE statement must be contained within a DO-group.
- A LEAVE statement must be in the same block as the DO statement to which it refers.
- A LEAVE statement label reference must refer to a label on a DO statement that heads a DO-group containing the LEAVE statement. The LEAVE statement must be in the same block as the labeled DO statement.
- The label reference specified with a LEAVE statement must be a label constant or a subscripted label constant with an integer constant subscript.

The following example shows a LEAVE statement with a label reference:

```
LOOP1: DO WHILE (MORE);
    .
    .
    .
    LOOP2: DO I = 1 TO 12;
    .
    .
    .
        IF QUAN(I) > 150 THEN LEAVE LOOP1;
    END;            /* loop 2 */
    .
    .
    .
END;                /* loop 1 */
```

In this example, the LEAVE statement transfers control to the first statement beyond the last END statement.

14.8 STOP Statement

The STOP statement terminates execution of the program. Its format is

```
STOP ;
```

The STOP statement terminates the program regardless of the current block activation, signals the FINISH condition, and closes all open files. If the main procedure has the RETURNS attribute, no return value is obtainable.

14.9 Null Statement

The null statement performs no action. Its format is

```
;
```

The null statement usually serves as the target statement of a THEN or ELSE clause in an IF statement, as the target of a WHEN or OTHERWISE clause in a SELECT statement, or as an action in an ON-unit. The following examples illustrate these uses.

```
IF A < B THEN GOTO COMPUTE;
     ELSE ;
```

In this example, no action takes place if A is greater than or equal to B; execution continues at the statement following ELSE ;. A construction of this type may be necessary when IF statements are nested (see Section 14.4).

```
SELECT;
    WHEN (condition A,B,C) GOTO FILE_READ;
    WHEN (condition D,E) GOTO UPDATE;
    OTHERWISE;
END;
```

In this example, control is passed to the next executable statement after END if conditions A, B, C, D, and E are not true.

```
ON ENDPAGE(SYSPRINT);
```

In this example, no action takes place upon execution of the ON-unit; the I/O operation that caused the ENDPAGE condition continues.

The null statement can also be used to declare two labels for the same executable statement, as in

```
LABEL1: ;
LABEL2: statement ...
```

Chapter 15

Error Handling

In PL/I, errors are signaled through ON conditions and handled by groups of statements called ON-units. An ON condition is any one of several named conditions whose occurrences during the execution of a program interrupt it. When an ON condition occurs, or is signaled, the corresponding ON-unit is executed.

This chapter describes the following statements, which allow you to establish and control ON-units:

- The ON statement—establishes an ON-unit to react to a particular ON condition
- The REVERT statement—cancels an ON-unit and returns control to a previously established ON-unit
- The SIGNAL statement—signals an ON condition explicitly
- The RESIGNAL built-in subroutine—within an ON-unit, signals the condition to another ON-unit

The description of the ON statement contains more information about ON-units and ON conditions.

15.1 ON Statement

The ON statement defines the action to be taken when a specific condition is signaled during the execution of a program.

The ON statement is an executable statement. For its ON-unit to be effective, it must be executed before the statement that signals the specified condition.

The format of the ON statement is

```
ON condition-name on-unit;
```

condition-name

The name of the specific condition for which an ON-unit is established. There is a keyword name associated with each condition. The conditions are summarized in Table 15–1 and treated in more detail in Section 15.1.4.

on-unit

The action to be taken when the specified condition is signaled. An ON-unit can be any single unlabeled statement except DECLARE, DO, END, ENTRY, FORMAT, IF, ON, PROCEDURE, RETURN, or SELECT. It can also be an unlabeled begin block.

If no ON-unit is established for a particular condition, the default PL/I condition handler, if any, is executed.

Table 15–1: Summary of ON Conditions

Condition Name	Usage
ANYCONDITION	Handles any condition not specifically handled by another ON-unit
ENDFILE	Handles end-of-file condition for a specified file
ENDPAGE	Handles end-of-page for a specified file with PRINT attribute
ERROR	Handles miscellaneous error conditions and conditions for which no specific ON-unit exists
FINISH	Handles program exit when the main procedure executes a RETURN statement, when any block executes a STOP statement, or when the program exits due to an error that is not handled by an ON-unit
FIXEDOVERFLOW	Handles fixed-point decimal and integer overflow exception conditions
KEY	Handles any error involving the key when using keyed access to a specified file
OVERFLOW	Handles floating-point overflow exception conditions
UNDEFINEDFILE	Handles any errors opening a specified file
UNDERFLOW	Handles floating-point underflow exception conditions
VAXCONDITION	Handles a specific VMS condition value
ZERODIVIDE	Handles divide-by-zero exception conditions

Subsequent sections provide the following information about ON-units:

- What an ON-unit can (and cannot) contain
- How PL/I searches for an ON-unit
- What happens when an ON-unit completes
- Descriptions of each ON condition
- Examples of ON-units

For the procedure that specifies OPTIONS (MAIN), PL/I defines a default condition-handling action that performs as follows:

- If the signal is the ENDPAGE condition, the default PL/I handler executes a PUT PAGE for the file, and then continues the program at the point at which ENDPAGE was signaled.
- If the signal is the ERROR condition and the severity is fatal, the default handler signals the FINISH condition. Then, one of the following occurs:
  - If a FINISH ON-unit is found, it is given a chance to execute. If it executes a nonlocal GOTO or signals another condition, program execution continues.
  - If no FINISH ON-unit is found, or if a FINISH ON-unit completes execution by handling the condition, then PL/I resignals the condition to the default VAX/VMS condition handler. This handler prints a message, displays a traceback, and terminates the program.
- If the signal is UNDERFLOW, a message is printed and execution continues. The value that caused the condition is replaced by zero. UNDERFLOW is signaled only if the procedure containing the expression specified OPTIONS (UNDERFLOW) on its PROCEDURE statement.
- If the signal is any condition other than ENDPAGE, ERROR with a fatal severity, or UNDERFLOW, the default PL/I handler signals the ERROR condition with the severity of the original condition. Then, one of the following occurs:
  - If an ERROR ON-unit is found, it is executed. If it completes execution by handling the condition, the program continues.
  - If an ERROR ON-unit is not found, the default PL/I handler resignals the condition. If this resignal results in return of control to the system, the default VAX/VMS condition handler prints a message and a traceback. If the error is a fatal error, the default handler terminates the program; otherwise, the program continues.

Note that PL/I does not provide any default condition handler unless a procedure in the program specifies OPTIONS (MAIN). If no PL/I-supplied default handler exists, conditions are handled by VAX/VMS default condition handlers.

If the default handler is not satisfactory, you can use an ON statement to establish your own ON-unit for a specific ON condition. The ON-unit is established following the execution of an ON statement that specifies that condition name. For example:

```
ON ENDFILE (ACCOUNTS) GOTO CLOSE_FILES;
```

This ON statement defines an ON-unit for an ENDFILE condition in the file specified by the name ACCOUNTS. The ON-unit consists of a single GOTO statement.

After an ON-unit is thus established, it remains in effect for the activation of the current block and all its dynamically descendent blocks, unless one of the following occurs:

- Another ON statement is specified for the same condition in a descendent block. The ON-unit established within the descendent block remains in effect as long as the descendent block is active.
- A REVERT statement is executed for the specified condition. A REVERT statement nullifies the most recent ON-unit for the specified condition.
- Another ON statement is specified for the same condition within the current block. Within the same block, an ON statement for a specific condition cancels the previous ON-unit.
- The block or procedure within which the ON-unit is established terminates. When a block exits, any ON-units it has established are canceled.

15.1.1 Contents of an ON-Unit

An ON-unit can consist of a single simple statement, a group of statements in a begin block, or a null statement. The following ON statement specifies a single statement in the ON-unit:

```
ON ERROR GOTO WRITE_ERROR_MESSAGE;
```

This ON statement specifies a GOTO statement that transfers control to the label WRITE_ERROR_MESSAGE in the event of the ERROR condition.

A simple statement must not be labeled and must not be any of the following:

DECLARE	FORMAT	PROCEDURE
DO	IF	RETURN
END	ON	SELECT
ENTRY		

An ON-unit can also consist of a sequence of statements in a begin block. For example:

```
ON ENDFILE (SYSIN) BEGIN;
   CLOSE FILE (TEMP);
   CALL PRINT_STATISTICS(TEMP);
   END;
```

This ON-unit consists of CLOSE and CALL statements that request special processing when the end-of-file condition occurs during reading of the default system input file, SYSIN.

When a BEGIN statement is specified for the ON-unit, it must not be labeled. The begin block can contain any statement except a RETURN statement.

A null statement specified for an ON-unit indicates that no processing is to occur when the condition occurs. Program execution continues as if the condition had been handled. For example:

```
ON ENDPAGE(SYSPRINT);
```

This ON-unit causes PL/I to continue output on a terminal regardless of the number of lines already output.

You can use the condition-handling built-in functions ONARGSLIST, ONCODE, ONFILE, and ONKEY to provide an ON-unit with information about the error that invoked it. Section 19.2 contains descriptions of these built-in functions.

15.1.2 Search for ON-Units

When a condition is signaled during the execution of a PL/I procedure, PL/I searches for an ON-unit to respond. It first searches the current block, that is, the block in which the condition occurs. If no ON-unit exists in this block for the specific condition, it searches the block that activated the current block, and then the block that activated that block, and so on.

PL/I executes the first ON-unit it finds, if any, that can handle the specified condition. If none is found, and if a procedure in the program specified OPTIONS (MAIN), the default PL/I condition handling is performed.

Figure 15–1 presents a program with ON-units established at several levels of block activation and shows the sequence for locating them.

15.1.3 Completion of ON-Units

PL/I executes an ON-unit as if it were a procedure having no parameters, that is, by creating a block activation for it and linking it to the block in which the condition occurred. The ON-unit can complete execution in any of the following ways:

- If the ON-unit executes a nonlocal GOTO statement, or if it invokes a subroutine or function that executes a nonlocal GOTO, program control is transferred to that statement and continues sequentially.
- If the ON-unit executes a STOP statement, then the FINISH condition is signaled. If no FINISH ON-unit exists, the program terminates.
- An ON-unit can use the RESIGNAL built-in subroutine to request that PL/I continue to search for an ON-unit to handle a specific condition.
- Normal completion of any ON-unit (except ERROR signaled as the default action) results in return of control either to the statement that caused the condition or to the statement immediately following the statement that caused the condition. However, the effects of normal return from ERROR, FIXEDOVERFLOW, OVERFLOW, UNDERFLOW, and ZERODIVIDE are generally unpredictable. Exceptions are cases of ERROR that are specifically documented to allow normal return, and ON-units that execute as a result of a SIGNAL statement. In the case of UNDERFLOW, return from the default PL/I condition handler continues execution, with zero as the result of the operation that caused the condition.

15.1.4 ON Condition Descriptions

The following paragraphs describe the ON conditions in alphabetic order.

ANYCONDITION Condition

The ANYCONDITION keyword designates an ON-unit established for all signaled conditions that are not handled by specific ON-units. It is not defined in the PL/I language, but is provided specifically for use in the VAX/VMS operating system environment. See the *VAX-11 PL/I User's Guide* for more information.

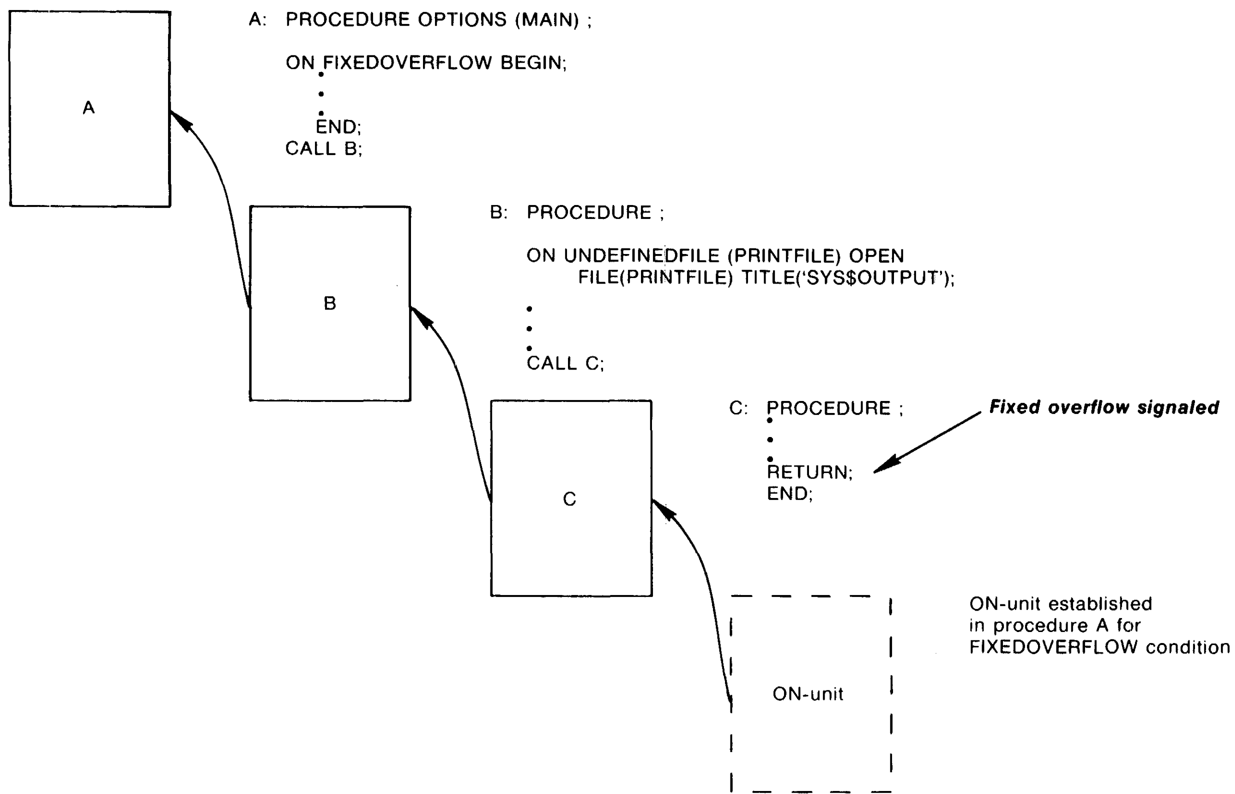


Figure 15-1: Search for ON-Units

ENDFILE Condition

The ENDFILE condition designates an end-of-file condition for a specific file. PL/I signals the ENDFILE condition when a GET or READ statement attempts an input operation on a file or device after the last data item has been input. The format of the ENDFILE condition name is

```
ENDFILE (file-reference)
```

file-reference

The name of a file constant or file variable for which the ENDFILE ON-unit is established.

An ENDFILE ON-unit can be established for any input file; the meaning of the end-of-file condition depends on the type of device. For example, it is signaled for a terminal device when a CTRL/Z is read. For a stream file, end-of-file is signaled whenever a GET statement attempts to access either an empty file or a file after its last input field has been read. For a record file, end-of-file is signaled when a READ statement is executed with the file at the end-of-file position, or when a read is attempted beyond the last record in the file.

An ON-unit established to handle end-of-file conditions can reference the ONFILE built-in function to determine the name of the file constant for which the condition was signaled.

If the ON-unit for the ENDFILE condition does not transfer control elsewhere in the program, control returns to the statement following the GET or READ statement that caused the condition to be signaled.

Once the ENDFILE condition is signaled, it remains in effect until the file is closed. Subsequent GET or READ statements for the file cause the ENDFILE condition to be signaled repeatedly.

ENDPAGE Condition

The ENDPAGE condition indicates when end-of-page is reached for a specific print file. The format of the ENDPAGE condition name is

```
ENDPAGE (file-reference)
```

file-reference

The name of the file constant or file variable for which the ENDPAGE ON-unit is to be established. The file must have the PRINT attribute.

The maximum number of lines that can be output on a single page is set by the PAGESIZE option of the OPEN statement. The maximum number of lines allowed on a single page is 32767. If not specified, PL/I uses the default page size.

PL/I signals the ENDPAGE condition when a PUT statement attempts to output a line beyond the last line specified for an output page. When the ENDPAGE condition is signaled, the current line number associated with the file is (pagesize+1). An ENDPAGE ON-unit allows you to provide special processing before output continues on a new page. For example:

```
ON ENDPAGE (PRINTFILE) BEGIN;
   PUT FILE (PRINTFILE) PAGE;
   PUT FILE (PRINTFILE) LIST(HEADER_LINE);
   PUT FILE (PRINTFILE) SKIP(2);
   END;
```

An ON-unit established to handle end-of-page conditions can reference the ONFILE built-in function to determine the name of the file constant for which the condition was signaled.

If the ON-unit does not transfer control elsewhere in the program, the line number is set to 1, and the program continues execution of the PUT statement. An ENDPAGE condition can occur only once per page of output. If the ON-unit does not specify a new page, then execution and output continue. The current line number can increase indefinitely; PL/I does not signal the ENDPAGE condition again. If, however, a LINE option on a PUT statement specifies a line number that is less than the current line, a new page is output and the current line is set to 1.

If the ENDPAGE condition is signaled during file processing, and no user-supplied ON-unit is present to handle the condition, PL/I starts output on a new page and continues processing. If the ENDPAGE condition is signaled as a result of a SIGNAL statement, the statement following the SIGNAL statement is executed and no page is output by default.

ERROR Condition

PL/I signals the ERROR condition in the following circumstances:

- When the compiled code or a PL/I routine detects an error for which no specific condition name exists in the VAX-11 PL/I language
- When the SIGNAL ERROR statement signals the condition
- When a default condition handler was established as a result of OPTIONS (MAIN), and a condition is signaled for which there is no corresponding ON-unit

When any condition is signaled for which no specific ON-unit is established, the default PL/I action for all conditions except ENDPAGE and UNDERFLOW is to signal the ERROR condition.

Any ON-unit being executed can reference the built-in function ONCODE, which returns the numeric condition value associated with the specific error that signaled the condition.

An ERROR ON-unit can act in the following ways:

- It can execute a nonlocal GOTO to resume normal program execution.
- It can pass control along to the default PL/I condition handler (or to an ERROR ON-unit established by a containing procedure) by calling the RESIGNAL built-in subroutine.
- It can terminate normally in cases that are specifically documented to allow normal return from an ERROR ON-unit. An example of such a case is a READ statement whose target is too small for the record it reads. In all other cases, the program continues, but execution is unpredictable.
- It can execute a STOP statement.

FINISH Condition

PL/I signals the FINISH condition in the following circumstances:

- When any procedure in the program executes the STOP statement
- When a procedure that specifies OPTIONS(MAIN) executes a RETURN statement, or, if the procedure does not execute a RETURN statement, when its corresponding END statement is executed
- When a program either exits as a result of a call to the system procedures SYS$EXIT or SYS$FORCEX (Force Exit), or after it is interrupted by an external CTRL key function
- When the SIGNAL FINISH statement signals the condition

If a FINISH ON-unit that executes as a result of a SIGNAL FINISH statement does not execute a nonlocal GOTO, control returns to the statement following SIGNAL FINISH. If the FINISH ON-unit executes as a result of any of the other three causes listed above, the program terminates.

FIXEDOVERFLOW Condition

The FIXEDOVERFLOW condition indicates that an overflow has occurred during an operation involving fixed-point numbers. PL/I signals the FIXEDOVERFLOW condition in the following circumstances:

- When the result of an arithmetic operation on a fixed-point decimal or binary integer value exceeds the maximum precision of the VAX-11 hardware. The maximum precision allowed for a fixed-point decimal or binary value is 31 (decimal or binary digits, respectively).
- When the source value of a fixed-point expression exceeds the declared precision of the target data type in assignment or conversion.

The value resulting from an operation that causes this condition is undefined.

Two VAX-11 hardware exceptions result in the FIXEDOVERFLOW condition. They are SS$_DECOVF (for a fixed-point decimal overflow) and SS$_INTOVF (for a fixed-point binary integer overflow). An ON-unit that receives control when FIXEDOVERFLOW is signaled can reference the ONCODE built-in function to determine which exception has occurred. To define an ON-unit to respond to either of these errors specifically, use the VAXCONDITION keyword.

If the ON-unit does not transfer control elsewhere in the program, control returns to the point at which the condition was signaled, but further execution is unpredictable.

KEY Condition

The KEY condition indicates a key error condition for a specific file. The format of the KEY condition name is

```
KEY (file-reference)
```

file-reference

A reference to the file constant or file variable for which the ON-unit is to be established.

PL/I signals the KEY condition during an operation on a keyed file when an error occurs in processing a key. Some examples of errors for which PL/I signals the KEY condition are

- The record indicated by the specified key cannot be found.
- The key specification required conversion from one data type to another, and the conversion is not valid.
- The key is not correctly specified.
- The key of a relative file exceeds the maximum record number specified when the file was created.

An ON-unit established to handle the KEY condition can obtain information about the condition by invoking the following built-in functions:

- The ONFILE built-in function, which returns the name of the file being processed when the condition was signaled
- The ONCODE built-in function, which returns the specific condition value associated with the error
- The ONKEY built-in function, which returns the key value that caused the condition to be signaled

If the ON-unit does not execute a nonlocal GOTO, control returns to the statement immediately following the statement that caused the KEY condition.

Section 16.5.2 contains an example of an ON-unit for the KEY condition.

OVERFLOW Condition

During an operation involving floating-point numbers, PL/I adjusts the exponent of a floating-point value, if possible, to represent the value with the specified precision. When the result of an arithmetic operation on a floating-point value exceeds the maximum allowed exponent size of the VAX–11 hardware, PL/I signals the OVERFLOW condition. The value resulting from an operation that causes this condition is undefined.

If the ON-unit does not execute a nonlocal GOTO, control returns to the point of the interruption, but execution is unpredictable.

UNDEFINEDFILE Condition

The UNDEFINEDFILE condition indicates that a specified file cannot be opened. The format of the UNDEFINEDFILE condition name is

```
UNDEFINEDFILE (file-reference)
```

file-reference

A reference to a file constant or file variable for which the ON-unit is established.

Some examples of errors that cause the UNDEFINEDFILE condition are

- The value specified by the TITLE option is an invalid file specification or the name of a file constant is not a valid VMS file specification, and no logical name assignment exists.
- The file is opened for input or update, and the specified file does not exist.
- An existing file is accessed with PL/I file description attributes that are inconsistent with the file's actual organization.
- Any system-detected file error prevents the file from being accessed.

The UNDEFINEDFILE condition lets you establish an ON-unit to provide processing when a file cannot be opened, for example, to provide a default file if no file is specified at run time. The ON-unit can obtain information about the condition by invoking the following built-in functions:

- The ONFILE built-in function, which returns the name of the file being processed when the condition was signaled
- The ONCODE built-in function, which returns the specific status value associated with the error

The action taken on a normal return from the UNDEFINEDFILE condition depends on whether the file was opened explicitly or implicitly:

- If the UNDEFINEDFILE condition was signaled following an explicit OPEN statement, then the normal action following the ON-unit execution is for the program to continue. If the ON-unit does not transfer

control elsewhere in the program, control returns to the statement following the OPEN statement.

- If the UNDEFINEDFILE condition was signaled during an implicit open attempt, the run-time system tests the state of the file. If it is not open, the ERROR condition is signaled. If the file was opened by the ON-unit, execution of the I/O statement continues.
- If an ON-unit receives control when an explicit OPEN results in the UNDEFINEDFILE condition, and the ON-unit does not handle the condition by opening the file or by transferring control elsewhere in the program, control returns to the statement following the OPEN. If an attempt is then made to access the file with an I/O statement, the UNDEFINEDFILE condition is signaled again when PL/I attempts the implicit open of the file. This time, PL/I signals the ERROR condition on completion of the ON-unit.

UNDERFLOW Condition

PL/I signals the UNDERFLOW condition when the absolute value of the result of an arithmetic operation on a floating-point value is smaller than the minimum value that the VAX-11 hardware can represent. The value resulting from an operation that causes this condition is set to zero. PL/I signals this condition only in procedures in which the UNDERFLOW option is enabled.

The UNDERFLOW option must be specified in each procedure for which underflow conditions are to be signaled. Following normal completion of the default PL/I action, control returns to the point of the interrupt, and execution continues with zero as the result of the operation. If a program-supplied ON-unit does not execute a nonlocal GOTO, control returns to the point of the interruption, but execution is unpredictable.

VAXCONDITION Condition

The VAXCONDITION condition name provides a way to signal and handle operating system or program-specific condition values. The format of the VAXCONDITION condition name is

```
VAXCONDITION (expression)
```

expression

An expression yielding a fixed binary value. It is evaluated when the ON statement is executed, not when the condition is signaled.

This condition name is provided specifically for PL/I procedures that interact with VAX/VMS operating system routines. For details on using the VAXCONDITION condition name and the meanings of system- and user-defined values you can specify, see the *VAX-11 PL/I User's Guide*.

ZERODIVIDE Condition

The ZERODIVIDE condition indicates a divide-by-zero operation. PL/I signals it when the divisor in a division operation has a value of zero. The value resulting from such an operation is undefined.

Following normal completion of the ON-unit, control returns to the point of the interruption, but execution is unpredictable.

15.1.5 ON-Unit Examples

The examples below illustrate some typical ON-units. The first example establishes an ON-unit for the FINISH condition. The ON-unit ensures that two files are closed properly, and calls a routine that stops a timer in an orderly fashion.

```
ON FINISH BEGIN;
     CLOSE FILE(INFILE);
     CLOSE FILE(OUTFILE);
     CALL TIMER_END;
     END;
```

Normally, the FINISH ON-unit should be declared in the main procedure; however, it will be executed on image exit if it is established in any block that is active when that occurs.

The next example contains an ERROR ON-unit that will terminate a program in an orderly fashion, should some error occur that is not handled by a specific ON-unit.

```
DECLARE STATUS FIXED BINARY(31);
   .
   .
   .
ON ERROR BEGIN;
    CLOSE FILE (INFILE);
    CLOSE FILE (OUTFILE);
    STATUS = ONCODE();
    GOTO FINIS;
    END;
   .
   .
   .
FINIS: RETURN (STATUS);
```

The ERROR ON-unit provides a cleanup procedure to ensure that the files identified as INFILE and OUTFILE are properly closed before the image exits. The ON-unit saves the value returned by ONCODE in the variable STATUS, and transfers control to a RETURN statement that returns the numeric value to the caller. If the procedure was invoked by a RUN command, this value is returned to the command interpreter, which in turn displays on the terminal the mnemonic code for the error and the error message.

The next example contains an ON-unit that changes the value of a bit variable when end-of-file is encountered.

```
DECLARE STATE_PTR POINTER,
        STATE_FILE FILE,
        EOF BIT(1) STATIC INIT('0'B);

ON ENDFILE(STATE_FILE) EOF = '1'B;

        OPEN FILE(STATE_FILE) INPUT SEQUENTIAL;
        READ FILE(STATE_FILE) SET(STATE_PTR);
        DO WHILE (^EOF);
            .
            .
            .
            READ FILE(STATE_FILE) SET(STATE_PTR);
            END;
```

The procedure reads the records in the file STATE_FILE until it encounters end-of-file. At that point, the ON-unit executes and changes the value of EOF from 0 to 1. This action causes the test in the DO WHILE statement to fail, terminating the loop that reads the records.

Section 16.5.2 demonstrates an ON-unit that handles errors encountered during record I/O operations.

15.2 REVERT Statement

The REVERT statement cancels an ON-unit established for a specified condition in the current block. Its format is

REVERT condition-name ;

condition-name

The condition name for which the ON-unit is to be reverted. It must be one of the condition names described in Section 15.1.4.

If no ON-unit is established for the specified condition for the current block, the REVERT statement has no effect. When the REVERT statement is executed for a specific condition for which an ON-unit exists, then:

- If a previous block activation specified an ON-unit for the indicated condition, that ON-unit will be executed when the condition is signaled.
- If no previous block activation specified an ON-unit for the condition, the default PL/I condition handling is reestablished.

15.3 SIGNAL Statement

The SIGNAL statement causes a specified condition to be signaled. Its format is

```
SIGNAL condition-name ;
```

condition-name

The condition to be signaled. It must be one of the condition names described in Section 15.1.4.

Most conditions occur as a result of a hardware trap or fault, or as a result of signaling by PL/I run-time procedures. The SIGNAL statement may be used within a program as a general-purpose communication technique. In particular, the VAXCONDITION keyword lets you specify unique program-defined values as well as operating-system-specific values.

15.4 RESIGNAL Built-In Subroutine

The RESIGNAL built-in subroutine is used in an ON-unit to "pass" a signaled condition, so that the run-time system will attempt to locate another ON-unit to handle the condition.

RESIGNAL works by setting up the internal mechanism for passing the signal. It does not, by itself, cause an exit from the ON-unit that calls it. Instead, it returns to the next statement in the ON-unit. Resignalling does not occur until execution of the ON-unit is completed.

The format of the RESIGNAL built-in subroutine is

```
CALL RESIGNAL();
```

When an ON-unit has determined that it cannot or should not respond to a condition, RESIGNAL permits the ON-unit to pass the signal along.

This subroutine is not part of the standard PL/I language. It is provided specifically for use in the VAX/VMS operating system environment.

Chapter 16

File Control

This chapter describes file specification syntax, statements, options, and subroutines of general use for controlling files, as well as techniques for detecting and responding to errors during file operations. The DECLARE statement with the FILE option declares a file constant or variable. The OPEN and CLOSE statements gain and terminate access to files. The ENVIRONMENT options, specified with the DECLARE and OPEN statements and (in some cases) with the CLOSE statement, control the attributes, processing, and disposition of a file. The file-handling built-in subroutines provide convenient ways of performing common operations. A final section on error handling describes how to detect an error, determine its nature, and take steps to correct it.

The subjects covered here are generally relevant to I/O operations on all types of files. The next two chapters describe the two broad categories of PL/I I/O, stream and record, as well as the statements, files, and file organization used with each.

16.1 File Control Statements

The DECLARE statement with the FILE attribute declares file constants and variables. The OPEN and CLOSE statements provide access to a file and terminate that access, respectively. They are applicable to both stream and record files.

16.1.1 Declaring a File

File constants and variables provide your program with access to files. Your program first declares a file constant or variable, and then associates the constant or variable with a file when it opens the file. (Section 16.1.2 contains information about opening files.)

A file declaration specifies an identifier, the FILE attribute, and one or more file description attributes that describe the type of I/O operation that

will be used to process the file. Subsequent I/O statements denote the file by a FILE option:

```
FILE(file-reference)
```

where file-reference is the name specified in the file's declaration. For example:

```
DECLARE INFILE FILE SEQUENTIAL INPUT;
OPEN FILE(INFILE);
```

Here, INFILE is the name of a file constant. A file constant is an identifier declared with the FILE attribute but not the VARIABLE attribute. Except for the default file constants SYSIN and SYSPRINT, all files must be declared before they can be opened and used.

By default, all file constants have the EXTERNAL attribute. Any external procedure that declares the identifier with the FILE attribute but not the INTERNAL attribute can access the same file constant and therefore the same physical file, as long as all other file attributes match.

You can also refer to files using file variables and file-valued functions. For example:

```
DECLARE ANYFILE FILE VARIABLE;

ANYFILE = INFILE;
OPEN FILE(ANYFILE);
```

The OPEN statement opens the file INFILE.

A file variable can also be given a value by passing a file constant as an argument or by return of a file constant as the value of a function. For example:

```
GETFILE: PROCEDURE (PRINTFILE);
DECLARE PRINTFILE FILE VARIABLE;
```

This file variable is given a value when the procedure GETFILE is invoked.

Table 16–1 lists the file description attributes that you can use in declarations with the FILE attribute. (You can also use these attributes with the OPEN statement, described in Section 16.1.2.) Section 16.1.2.1 and Chapters 17 and 18 contain more information about these attributes and the operations they denote.

16.1.2 OPEN Statement

The OPEN statement explicitly opens a PL/I file with a specified set of attributes that describe the file and the method for accessing it. The format of the OPEN statement is

```
OPEN FILE(file-reference)
        [file-description-attribute ...] ;
```

file-reference

A reference to the file to be opened. If the file is already open, the OPEN statement has no effect.

file-description-attribute ...

The attributes of the file. They are merged with any permanent attributes of the file specified in its declaration. Default rules are then applied to the union of these sets of attributes to complete the set of attributes in effect for this opening. (Section 16.1.2.2 describes this process.)

The attributes and options you can specify on the OPEN statement are

DIRECT	PRINT
ENVIRONMENT(option,...)	RECORD
INPUT	SEQUENTIAL
KEYED	STREAM
LINESIZE(expression)	TITLE(expression)
OUTPUT	UPDATE
PAGESIZE(expression)	

INPUT, OUTPUT, and TITLE are described in Section 16.1.2.1. ENVIRONMENT and its options are described in Section 16.3. LINESIZE, PAGESIZE, PRINT, and STREAM are described at the beginning of Chapter 17. DIRECT, KEYED, RECORD, SEQUENTIAL, and UPDATE are described at the beginning of Chapter 18.

Table 16-1: File Description Attributes

Attribute	Meaning
DIRECT	Records in the file will be accessed randomly only.
ENVIRONMENT	Specifies VAX/VMS-specific file properties.
INPUT	The file is an input file and will only be read.
KEYED	Records in the file will be accessed by key.
OUTPUT	The file is an output file and will only be written.
PRINT	The format of the file will be suitable for output on a printer or terminal.
RECORD	The file will be accessed using record I/O statements.
SEQUENTIAL	Records in the file will be accessed sequentially.
STREAM	The file will be accessed using stream I/O statements.
UPDATE	The file will be accessed for both reading and writing, and records may be rewritten and deleted.

The following examples demonstrate the OPEN statement.

```
DECLARE INFILE FILE;

OPEN FILE (INFILE);
```

Neither INFILE's declaration nor its open specify any file description attributes. PL/I applies the default attributes STREAM and INPUT. If any statement other than GET is used to process this file, the ERROR condition is signaled.

```
DECLARE STATE_FILE FILE KEYED;

OPEN FILE(STATE_FILE) UPDATE;
    .
    .
    .
CLOSE FILE(STATE_FILE);
OPEN FILE(STATE_FILE) INPUT SEQUENTIAL;
```

The file STATE_FILE is declared with the KEYED attribute, which implies the RECORD attribute. On the first OPEN statement that specifies this file, the file is given the UPDATE attribute and opened for updating; that is, READ, WRITE, REWRITE, and DELETE statements may be used to operate on records in the file.

The second OPEN statement specifies the INPUT and SEQUENTIAL attributes. During this opening, the file may be accessed by sequential and keyed READ statements; REWRITE, DELETE, and WRITE statements may not be used.

```
DECLARE COPYFILE FILE OUTPUT;
OPEN FILE(COPYFILE) TITLE('COPYFILE.DAT');
```

The file associated with the file constant COPYFILE is opened for output. Each time this program is run, it creates a new version of the stream file COPYFILE.DAT.

16.1.2.1 General-Purpose Attributes and Options

The INPUT and OUTPUT attributes and the TITLE option are equally applicable to stream and record files. They are described below.

INPUT Attribute

The INPUT file description attribute indicates that the associated file is to be an input file; that is, it represents an external source of data. Specify the INPUT attribute on a DECLARE statement for a file constant or on an OPEN statement to access the file for reading. For a stream file, INPUT indicates that the file will be accessed using GET statements; for a record file, INPUT indicates that it will be accessed using only READ statements.

The INPUT attribute conflicts with the OUTPUT, UPDATE, and PRINT attributes and with any data type attribute other than FILE.

OUTPUT Attribute

The OUTPUT file description attribute indicates that data is to be written to, and not read from, the associated external device or file. Specify the OUTPUT attribute on a DECLARE statement for a file constant or on an OPEN statement to access the file for writing. For a stream file, OUTPUT indicates that the file will be accessed using PUT statements; for a record file, OUTPUT indicates that it will be accessed using only WRITE statements.

The OUTPUT attribute conflicts with the INPUT and UPDATE attributes and with any data type attributes other than FILE.

TITLE Option

The TITLE option is specified in an OPEN statement to designate the external file specification of the file to be associated with the PL/I file. The TITLE option can be specified only on the OPEN statement for a file, not on the DECLARE statement. Its format is

```
TITLE(expression)
```

expression

A character-string expression of up to 128 characters, representing an external file specification for the file.

The file specification can be any valid VAX/VMS file specification, device name, or logical name. When the name given with TITLE does not fully specify a VAX/VMS file or device, VAX-11 PL/I

1. Performs logical name translation.
2. Applies default values given in the DEFAULT_FILE_NAME option of the ENVIRONMENT attribute.
3. Applies system defaults.

This process is described in more detail in Section 16.2.

16.1.2.2 Opening a File

A file is opened explicitly by an OPEN statement or implicitly by a READ, WRITE, REWRITE, DELETE, PUT, or GET statement issued for (or built-in subroutine reference to) a file that is not open. Opening a file either explicitly or implicitly in PL/I has the following effects:

- Any permanent attributes specified in the DECLARE statement of a file constant are merged with any attributes specified in the OPEN statement, or with the attributes implied by the context of the opening. (For example, if no attributes are specified for a file in its declaration, and the first reference to it is a GET statement, PL/I opens the file with the INPUT and STREAM attributes.)

- The merged attributes apply to the file for the duration of this opening only. When the file is closed, only its permanent attributes are retained for subsequent openings.
- The file specification is determined, using the value of the TITLE option. (Section 16.2.1 describes this process.)
- If the file already exists, it is located and its attributes are checked for compatibility with those specified or implied by the open.
- If the file does not exist, and if the attempted access does not require that it exist, PL/I creates a new file using the specified or implied attributes to determine its organization.
- If the open is successful, the file is positioned.

Some of these steps are described in more detail below. If an error occurs during the opening of a file, the UNDEFINEDFILE condition is signaled.

Establishing the File's Attributes

The file description attributes specified by the opening context are merged with the file's permanent attributes. Duplicate specification of an attribute is allowed only for one that does not specify a value.

If the set of attributes is not complete, it is augmented with implied attributes. Table 16-2 summarizes the attributes that may be added to an incomplete set.

Table 16-2: File Description Attributes Implied at Open Time

Attribute	Implied Attributes
DIRECT	RECORD KEYED
KEYED	RECORD
PRINT	STREAM OUTPUT
SEQUENTIAL	RECORD
UPDATE	RECORD

If the set of attributes is still not complete, PL/I takes the following steps:

1. If neither STREAM nor RECORD is present, STREAM is supplied.
2. If neither INPUT, OUTPUT, nor UPDATE is present, INPUT is supplied.
3. If RECORD is specified, but neither SEQUENTIAL nor DIRECT is present, SEQUENTIAL is supplied.

4. If the file's identifier is associated with SYSPRINT, and the attributes STREAM and OUTPUT are present, PRINT is supplied.
5. If the set contains the LINESIZE option, it must contain STREAM and OUTPUT. If it contains these attributes and does not contain LINESIZE, the default system line size value is supplied.
6. If the set contains the PAGESIZE option, it must contain PRINT. If PRINT is present but PAGESIZE is not, the default PL/I page size is supplied.
7. If the set does not contain TITLE, a default option TITLE(name) is supplied, where name is the name of the file constant associated with the file.

The completed set of attributes applies only for the current opening of the file. The file's permanent attributes, specified in the declaration of the file, are not changed.

Accessing an Existing File

An open accesses an existing file if the file specified by the TITLE option actually exists and if the following attributes are present:

- The file is opened for INPUT or UPDATE.
- The file is opened with the OUTPUT attribute and with the ENVIRONMENT (APPEND) option.

Whenever PL/I accesses an existing file, its organization is checked for compatibility with the PL/I attributes specified. If any incompatibilities exist, the UNDEFINEDFILE condition is signaled.

Creating a File

An open creates a new file if the following are all true:

- The OUTPUT attribute is specified.
- The TITLE option, after logical name translation and the application of system defaults, specifies a mass storage device, for example, a disk or a tape.
- The ENVIRONMENT(APPEND) option is not specified.

ENVIRONMENT options can specify the organization and record format of a new file. If no ENVIRONMENT options are given, the new file's organization is determined as follows:

- If the KEYED attribute is present, PL/I creates a relative file with a maximum record size of 480 bytes and a maximum record number of 0.

- If the PRINT attribute is present, PL/I creates a sequential file having variable-length records, a maximum record length equal (in bytes) to the line size (see Section 17.2.1), and a fixed-control field used by PL/I to store carriage control information.
- If neither KEYED nor PRINT is specified, PL/I creates a sequential file with variable-length records and a maximum record size of 510.

When you open a file with the RECORD and OUTPUT attributes, you may use only WRITE statements to access the file. If the file has the KEYED attribute as well, the WRITE statements must include the KEYFROM option.

File Positioning

When PL/I opens a file, the initial positioning depends on the type of file (record or stream), the access mode, and certain ENVIRONMENT options. Section 17.2 describes file positioning for stream files; Section 18.1, for record files.

16.1.3 CLOSE Statement

The CLOSE statement dissociates a PL/I file from the physical file with which it was associated when it was opened. The format of the CLOSE statement is

```
CLOSE FILE(file-reference) [ENVIRONMENT(option,...)];
```

file-reference

The file to be closed. If it is already closed, the CLOSE statement has no effect.

ENVIRONMENT(option,...)

One or more of the ENVIRONMENT options listed below, separated by commas:

BATCH	SPOOL
DELETE	TRUNCATE
REWIND_ON_CLOSE	

No other ENVIRONMENT options are valid. They are summarized in Section 16.3.

The following examples illustrate the use of the CLOSE statement.

```
CLOSE FILE(INFILE);
```

This CLOSE statement closes the file constant INFILE.

```
DECLARE STATE_FILE FILE KEYED;

OPEN FILE(STATE_FILE) DIRECT UPDATE;
    .
    .
    .
CLOSE FILE(STATE_FILE);
OPEN FILE(STATE_FILE) INPUT SEQUENTIAL;
```

The file STATE_FILE is declared with the KEYED attribute. On the first OPEN statement that specifies this file, the file is given the DIRECT and UPDATE attributes and opened for updating; that is, it can be accessed only by key. The CLOSE statement closes the file, and the second OPEN statement specifies the INPUT and SEQUENTIAL attributes. The file may then be accessed sequentially.

16.2 PL/I Files and VAX/VMS File Specifications

In a PL/I program, all I/O operations are performed on a file, using the name of a file constant or file variable. When the file is opened, PL/I associates the name of the file constant with a specific device or file on the computer system.

When a file variable is specified in an OPEN statement or in an I/O statement, the name used is that of the file constant with which the variable is currently associated. For example:

```
DECLARE F FILE,
        G FILE VARIABLE;

        G = F;
        OPEN FILE(G);
```

In this example, F is a file constant and G is a file variable assigned the value of F. In the OPEN statement, PL/I uses the name F to associate the PL/I file with a VAX/VMS file. The default file would be F.DAT.

The sections that follow describe in more detail how VAX-11 PL/I associates a file constant with a device or file.

16.2.1 The TITLE Option

When you specify the TITLE option on an OPEN statement, you can include all or part of a VAX/VMS file specification to indicate the file or device to be associated with the PL/I file. The following examples illustrate the use of the TITLE option.

```
OPEN FILE (OUTFILE)
     TITLE('DB1:[PAYROLL.DAT]JANUARY.LOG;2');
```

This file specification completely defines a file on the local VAX/VMS system.

```
OPEN FILE (PRINTFILE) PRINT TITLE('LPC0:SAMPLE.DAT');
```

This output file will be directed to the system printer device named LPC0:; the listing file will have the title SAMPLE.DAT on its burst page. VAX/VMS spools low-speed I/O devices such as printers by accumulating data for the device in a file, and then queueing the file for processing when it is closed.

```
NAME = 'TEST'||COUNT;
    .
    .
    .
OPEN FILE(NEWFILE) OUTPUT TITLE(NAME);
```

The specification of this file is determined by the value of COUNT. For example, if COUNT is 5 when this OPEN statement executes, the file created is TEST5.DAT.

When no TITLE option is specified, PL/I supplies a default value for the file's title. The default title is the name of the file constant associated with the PL/I file. Whenever a title does not completely specify a file, VAX-11 PL/I takes the following steps, in order:

1. It performs logical name translation. If there is a colon (:) present in the TITLE option, the file system attempts to find an equivalence name for the portion of the file specification on the left of the colon. If there are no punctuation marks in the TITLE option, the file system attempts to find an equivalence name for the entire specification.
2. It supplies missing fields from the value specified in the DEFAULT_FILE_NAME option of the ENVIRONMENT attribute, if that option is specified.
3. It then applies system defaults to complete the file specification.

If the file specification that is finally achieved is invalid (for example, it contains a dollar sign or underscore character) or represents an illegal device or file (for example, an input file cannot be found), the UNDEFINEDFILE condition is signaled.

16.2.2 Using Logical Names

At the command level before executing a program, you can create a logical name to assign a VAX/VMS file specification to the identifier of a PL/I file

constant or to a value specified in a TITLE option. For example, suppose your PL/I program declares and opens a file as follows:

```
DECLARE INFILE FILE;
    .
    .
    .
OPEN FILE (INFILE) RECORD INPUT;
```

Before running the program, you might associate a VAX/VMS file with the identifier INFILE:

```
$ DEFINE  INFILE  DB1:[TEMP]A.DAT
```

The DEFINE command gives the PL/I file INFILE the VAX/VMS file equivalent of DB1:[TEMP]A.DAT. In VAX/VMS terms, the name INFILE is a logical name, and the name DB1:[TEMP]A.DAT is an equivalence name for the logical name.

You can also use the DEFINE command to specify alternate device or file equivalents for the PL/I default file constants SYSIN and SYSPRINT. For example, to redirect output for the default file SYSPRINT, you could specify a command as follows:

```
$ DEFINE SYSPRINT TEST.OUT
```

While this assignment is in effect, any PL/I procedure that outputs data to SYSPRINT (without opening SYSPRINT with an explicit title) will create a file named TEST.OUT on the current default device.

Logical names may also be established by other commands. For example, you can specify a logical name for a device when you enter an ALLOCATE or MOUNT command while placing the device on line. For example:

```
$ ALLOCATE
$_Device:   MT:
$_Log_Name: INFILE
  _MTA1: ALLOCATED
```

This ALLOCATE command allocates a tape drive and establishes the logical name INFILE for it. When a PL/I program reads from the file INFILE, the system will translate the name INFILE and use the tape MTA1: as the input device.

16.2.3 Process Permanent Logical Names

The system provides every user and every batch job with a default set of process logical name assignments, which are listed in Table 16–3. Because the files associated with these assignments exist for the life of the process, or job, and because they are permanently open, they are called process permanent files.

Table 16-3: Default Process Logical Names

Logical Name	Default Equivalence Name
SYS$INPUT	Input stream. For an interactive user, this is the terminal or a command procedure file; for a batch job, the input command file.
SYS$OUTPUT	Output stream. For an interactive user, this is the terminal; for a batch job, the batch job log file.
SYS$ERROR	Error stream. Unless overridden by the user, it is the same as SYS$OUTPUT.
SYS$DISK	Default device and directory.
SYS$COMMAND	Default command stream. For an interactive user, this is the terminal; for a batch job, the batch job input command file.

The default files associated with the GET and PUT statements, SYSIN and SYSPRINT, are defined by PL/I as follows:

Statement	Default PL/I File	Default TITLE
GET	SYSIN	SYS$INPUT
PUT	SYSPRINT	SYS$OUTPUT

Thus, when your program executes a GET statement that does not specify the FILE option, and if SYSIN was not explicitly opened with a title, the run-time system and the file system perform the following translations:

1. PL/I attempts to translate the logical name SYSIN. If no logical name assignment exists for it, PL/I replaces the name SYSIN with the name SYS$INPUT.
2. The system translates the logical name SYS$INPUT. The resulting file specification is your current input device.

A similar set of associations occurs when a program executes a PUT statement without the FILE option: the resulting output is written to the current output file, SYS$OUTPUT.

16.2.4 Expanding File Specifications

After logical name translation, the defaults that the VAX-11 PL/I I/O system applies are as follows:

Field	System Default Provided
node	Local system
device	Current default device
directory	Current default directory
file name	None

Field	System Default Provided
file type	DAT
version number	For an input file, the most recent version; for an output file, the highest existing version number plus 1

16.3 Summary of ENVIRONMENT Options

The options to the ENVIRONMENT attribute provided by VAX-11 PL/I let you

- Describe the attributes of a file when it is created.
- Request special processing and optimization options when the file is being read or written.
- Specify the disposition of a file when it is closed.

The options to the PL/I ENVIRONMENT attribute are summarized in alphabetical order in Table 16-4. The columns in the table provide the following information:

Option	Gives the name of the ENVIRONMENT option and its argument, if any. An option that does not show an argument may be specified with a Boolean argument.
Usage	Gives a brief description of the option.
Specify at	Specifies when the option is meaningful. The possible items in this column are Create—the option can be specified on a DECLARE or OPEN. It is meaningful only when a file is created. Open—the option can be specified on a DECLARE or OPEN. It is meaningful when an existing file is opened. Close—the option can be specified on a DECLARE, OPEN, or CLOSE. It takes effect when the file is closed. Update—the option is meaningful when an existing file is opened with the UPDATE attribute or with the ENVIRONMENT option APPEND.
Valid I/O types	Indicates whether the option is valid for stream or record files.
Default Value	Indicates the default value, if any, when the option is not specified for a file.
Data Type	Specifies the required data type of the argument.

The *VAX-11 PL/I User's Guide* contains complete information and examples for all the ENVIRONMENT options.

Table 16-4: Summary of ENVIRONMENT Options

Option	Usage	Specify At	Valid I/O Types	Default Value	Data Type
APPEND	Places output for a file at the end of a file.	Create Open	Record Stream	Disabled	BIT(1)
BATCH	Submits a copy of the file to the system batch job queue on close.	Create Open Close	Record Stream	Disabled	BIT(1)
BLOCK_BOUNDARY_FORMAT	Indicates that records must not cross block boundaries.	Create	Record Stream	Disabled	BIT(1)
BLOCK_IO	Specifies a file will be read or written by blocks instead of records.	Create Open	Record	Disabled	BIT(1)
BLOCK_SIZE(expression)	Specifies the size of a block for the creation of a magnetic tape file.	Create	Record Stream	Mount value	FIXED BIN(31)
BUCKET_SIZE(expression)	Defines the number of 512-byte blocks in a bucket for an indexed sequential or a relative file.	Create	Record	Maximum record size	FIXED BIN(31)
CARRIAGE_RETURN_FORMAT	Indicates that records in the file will be printed with default carriage control.	Create	Record	Enabled	BIT(1)
CONTIGUOUS	Specifies that an output file must be placed in a physically contiguous extent on the disk.	Create	Record Stream	Disabled	BIT(1)
CONTIGUOUS_BEST_TRY	Requests that if possible an output file be placed in a physically contiguous extent on the disk.	Create	Record Stream	Disabled	BIT(1)

Table 16-4 (Cont.): Summary of ENVIRONMENT Options

Option	Usage	Specify At	Valid I/O Types	Default Value	Data Type
CREATION_DATE(variable)	Overrides default creation data of file.	Create	Record Stream	Current date and time	BIT(64) ALIGNED
CURRENT_POSITION	Leaves magnetic tape positioned at last close.	Create Open	Record Stream	Disabled	BIT(1)
DEFAULT_FILE_NAME (expression)	Defines a default file specification for a file.	Create Open	Record Stream	'.DAT'	CHAR(128)
DEFERRED_WRITE	Requests file system optimization of output.	Create Open	Record	Disabled	BIT(1)
DELETE	Specifies that the file be deleted when it is closed.	Create Open Close	Record Stream	Disabled	BIT(1)
EXPIRATION_DATE(variable)	Defines the expiration date for a magnetic tape file.	Create	Record Stream	Creation date	BIT(64) ALIGNED
EXTENSION_SIZE(expression)	Specifies a default extension size for a disk file.	Create Open	Record Stream	System default	FIXED BIN(31)
FILE_ID(variable) FILE_ID_TO(variable)	Identifies a file by its internal file identification.	Create Open	Record Stream	n/a n/a	(6) FIXED BIN(31) (6) FIXED BIN(31)
FILE_SIZE(expression)	Defines the initial number of blocks to allocate for a file.	Create	Record Stream	n/a	FIXED BIN(31)

Table 16-4 (Cont.): Summary of ENVIRONMENT Options

Option	Usage	Specify At	Valid I/O Types	Default Value	Data Type
FIXED_CONTROL_SIZE (expression) FIXED_CONTROL_SIZE_TO (variable)	Defines records as variable length with fixed-length control and specifies the size of the fixed control area. On open, returns the length of the fixed control area.	Create Open	Record	Disabled	FIXED BIN(31)
FIXED_LENGTH_RECORDS	Specifies a file with fixed-length records of a maximum record size.	Create	Record	Disabled	BIT(1)
GROUP_PROTECTION (expression)	Defines the type of file access allowed to members of the owner's group.	Create	Record Stream	Current process default	CHAR(4)
IGNORE_LINE_MARKS	Specifies that end-of-line characters are not to be treated as field delimiters in GET LIST statements.	Create Open	Stream	Disabled	BIT(1)
INDEX_NUMBER(expression)	Specifies the initial index to use in accessing records in an indexed sequential file.	Create Open	Record	0	FIXED BIN(31)
INDEXED	Defines an indexed sequential file.	Create Open	Record	Disabled	BIT(1)
INITIAL_FILL	Requests the file system to leave unused space in file index overflow buckets.	Open	Record	Disabled	BIT(1)
MAXIMUM_RECORD_NUMBER (expression)	Specifies the largest record number that will be valid for records in a relative file.	Create	Record	0	FIXED BIN(31)

Table 16-4 (Cont.): Summary of ENVIRONMENT Options

Option	Usage	Specify At	Valid I/O Types	Default Value	Data Type
MAXIMUM_RECORD_SIZE (expression)	Specifies the maximum size that is valid for any record in the file.	Create	Record	512 bytes[1]	FIXED BIN(31)
MULTIBLOCK_COUNT (expression)	Specifies the number of blocks to allocate for file system buffering.	Create Open	Record	Current process default	FIXED BIN(31)
MULTIBUFFER_COUNT (expression)	Specifies the number of buffers to allocate for file system buffering.	Create Open	Record	Current process default	FIXED BIN(31)
NO_SHARE	Prohibits all type of shared access to the file.	Create Open	Record	[2]	BIT(1)
OWNER_GROUP(expression)	Specifies the group number in the user identification code (UIC) of the owner of the file.	Create	Record Stream	Current process group number	FIXED BIN(31)
OWNER_MEMBER(expression)	Specifies the member number in the user identification code (UIC) of the owner of the file.	Create	Record Stream	Current process member number	FIXED BIN(31)
OWNER_PROTECTION (expression)	Specifies the type of file access allowed the owner of the file.	Create	Record Stream	Current process default	CHAR(4)

1. For sequential files with fixed-length records. For sequential files with variable-length records, the default is 510 bytes. For relative files, the default is 480 bytes.
2. Disabled if the file is opened for input, enabled if opened for output or update.

Table 16-4 (Cont.): Summary of ENVIRONMENT Options

Option	Usage	Specify At	Valid I/O Types	Default Value	Data Type
SHARED_READ	Allows other users to read records in the file.	Create Open	Record	[3]	BIT(1)
SHARED_WRITE	Allows other users to read and write records in the file.	Create Open	Record	Disabled	BIT(1)
SPOOL	Queues a copy of the file to the system printer when the file is closed.	Create Open Close	Record Stream	Disabled	BIT(1)
SUPERSEDE	Replaces an existing file with the same file name, file type, and version number.	Create	Record Stream	Disabled	BIT(1)
SYSTEM_PROTECTION (expression)	Defines the type of file access allowed to users with system user identification codes.	Create	Record Stream	Current process default	CHAR(4)
TEMPORARY	Specifies a temporary file for which no directory entry is made.	Create	Record Stream	Disabled	BIT(1)
TRUNCATE	Truncates a sequential file at its logical end-of-file when it is closed.	Create Update Close	Record Stream	Disabled	BIT(1)

3. Enabled if the file is opened for input, otherwise disabled.

Table 16-4 (Cont.): Summary of ENVIRONMENT Options

Option	Usage	Specify At	Valid I/O Types	Default Value	Data Type
PRINTER_FORMAT	Specifies that records in the file will be printed using printer format carriage control embedded in the fixed control area of the records.	Create	Record	Disabled	BIT(1)
READ_AHEAD	Requests file system optimization on read operations.	Open	Record Stream	Enabled	BIT(1)
READ_CHECK	Requests verification of read operations.	Create Open	Record Stream	Disabled	BIT(1)
RECORD_ID_ACCESS	Indicates that records will be accessed by internal file system identification.	Create Open	Record	Disabled	BIT(1)
RETRIEVAL_POINTERS (expression)	Specifies the number of file system extent pointers to maintain for file access.	Create Open	Record Stream	Current system default	FIXED BIN(31)
REWIND_ON_CLOSE	Requests that a magnetic tape volume be rewound when the file is closed.	Create Open Close	Record Stream	Disabled	BIT(1)
REWIND_ON_OPEN	Requests that a magnetic tape volume be rewound when the file is opened.	Create Open	Record Stream	Enabled	BIT(1)
SCALARVARYING	Specifies that varying character strings will be read/written using the entire storage of the variable.	Create Open	Record	Disabled	BIT(1)

Table 16-4 (Cont.): Summary of ENVIRONMENT Options

Option	Usage	Specify At	Valid I/O Types	Default Value	Data Type
WORLD_PROTECTION (expression)	Specifies the type of file access allowed to general system users.	Create	Record Stream	Current process default	CHAR(4)
WRITE_BEHIND	Requests file system optimization on output operations.	Create Update	Record Stream	Disabled	BIT(1)
WRITE_CHECK	Requests verification of output operations.	Create Update	Record Stream	Disabled	BIT(1)

16.4 Summary of File-Handling Built-In Subroutines

In addition to the PL/I input and output statements and the functions and features available through the options of the ENVIRONMENT attribute, there are also several built-in file-handling subroutines. These subroutines invoke VAX-11 RMS procedures. They are "builtin" because you do not need to declare them before using them in a PL/I program. These subroutines are summarized in Table 16–5, and described in the sections that follow.

Table 16–5: Summary of File-Handling Built-in Subroutines

Subroutine	Function
DISPLAY	Returns information about a file
EXTEND	Allocates additional disk blocks for a file
FLUSH	Requests the file system to write all buffers onto disk to preserve the current status of a file
NEXT_VOLUME	Begins processing the next volume in a multivolume tape set
REWIND	Positions a file at its beginning or at a specific record
SPACEBLOCK	Positions a file forward or backward a specified number of blocks

16.4.1 DISPLAY Built-In Subroutine

The DISPLAY built-in subroutine returns information about a specified file. Its calling sequence is

```
CALL DISPLAY (file-reference,variable-reference) ;
```

file-reference

Specifies the file variable or constant for which information is to be obtained. If the file is not currently open, the DISPLAY subroutine implicitly opens it with the attributes specified in the file's declaration.

variable-reference

Specifies the name of a structure variable in which information about the file is to be placed.

The format of the data returned by DISPLAY is defined in the data structure PLI_FILE_DISPLAY. This structure is declared in the text module PLI_FILE_DISPLAY in the default INCLUDE library PLISYSDEF (the PL/I compiler searches this library by default when it compiles a PL/I program). Each member of PLI_FILE_DISPLAY contains, on return

from a call to DISPLAY, a value associated with the file for which information is requested. To reference a value, you need only refer to the corresponding member name in the structure. Tables 16–6 through 16–8 summarize the members of the structure in the following categories:

- Members containing information about the settings of ENVIRONMENT options (listed in Table 16–6)
- Members containing information on file attributes (listed in Table 16–7)
- Members containing information on device attributes (listed in Table 16–8)

The structure PLI_FILE_DISPLAY is declared with the BASED attribute; thus, to use this variable, you must also declare a pointer variable to reference the structure and use an ALLOCATE statement to allocate storage for it before calling DISPLAY. For example:

```
%INCLUDE PLI_FILE_DISPLAY;
DECLARE STATE_FILE FILE RECORD KEYED,
        FILEPTR POINTER;
        OPEN FILE(STATE_FILE);
        ALLOCATE PLI_FILE_DISPLAY SET (FILEPTR);
        CALL DISPLAY (STATE_FILE,
                      FILEPTR->PLI_FILE_DISPLAY);
```

Following this call to DISPLAY, you can reference any of the members of FILEPTR->PLI_FILE_DISPLAY to determine information about the file STATE_FILE. The next statements use the EXPANDED_TITLE field to display the expanded file specification of STATE_FILE, and the INDEXED and NUMBER_OF_KEYS fields to display the number of keys in the file:

```
PUT SKIP EDIT ('File',FILEPTR->EXPANDED_TITLE,
              'opened for input')
              (A,X,A,X,A);
IF FILEPTR->INDEXED THEN PUT SKIP EDIT
      ('It is indexed with',
           FILEPTR->NUMBER_OF_KEYS,'Keys')
      (A,X,F(3),X,A);
```

If you do not use the structure PLI_FILE_DISPLAY, as shown in the example, you must provide a structure with the same declaration. To obtain a copy of PLI_FILE_DISPLAY, use the LIBRARY command, as in this example:

```
$ LIBRARY/TEXT/EXTRACT=PLI_FILE_DISPLAY-
$_/OUTPUT=FILESTRUC.PLI SYS$LIBRARY:PLISYSDEF
```

where FILESTRUC.PLI is the name of the output file into which the LIBRARY command will copy PLI_FILE_DISPLAY.

Table 16–6 summarizes the values returned by DISPLAY that correspond to ENVIRONMENT options and the data type of each structure member.

Table 16-6: ENVIRONMENT Option Values Returned by DISPLAY

Member Name	Data Type of Value Returned	Meaning
APPEND	BIT(1)	APPEND option is enabled/disabled
BATCH	BIT(1)	BATCH option is enabled/disabled
BLOCK_BOUNDARY_FORMAT	BIT(1)	Records cannot cross block boundaries
BLOCK_IO	BIT(1)	File is opened for block I/O
BLOCK_SIZE	FIXED BIN(31)	Block size of file (magtape files only)
BUCKET_SIZE	FIXED BIN(31)	Bucket size of file (disk files only)
CARRIAGE_RETURN_FORMAT	BIT(1)	Records have carriage return carriage control
CONTIGUOUS	BIT(1)	CONTIGUOUS option is enabled/disabled
CONTIGUOUS_BEST_TRY	BIT(1)	CONTIGUOUS_BEST_TRY option is enabled/disabled
CREATION_DATE	BIT(64)	Creation date of file
CURRENT_POSITION	BIT(1)	CURRENT_POSITION option is enabled/disabled
DEFERRED_WRITE	BIT(1)	DEFERRED_WRITE option is enabled/disabled
DELETE	BIT(1)	DELETE option is enabled/disabled
EXPIRATION_DATE	BIT(64)	Expiration date (magnetic tape files only)
EXTENSION_SIZE	FIXED BIN(31)	Current extension size (disk files only)
FILE_ID(6)	FIXED BIN(31)	File identification (disk files only)
FILE_SIZE	FIXED BIN(31)	File allocation (disk files only)
FIXED_CONTROL_SIZE	FIXED BIN(31)	Size of fixed control area

Table 16-6 (Cont.): ENVIRONMENT Option Values Returned by DISPLAY

Member Name	Data Type of Value Returned	Meaning
FIXED_LENGTH_RECORDS	BIT(1)	File has fixed-length records
GROUP_PROTECTION	CHAR(4) VARYING	Protection for group members
IGNORE_LINE_MARKS	BIT(1)	IGNORE_LINE_MARKS option is enabled/disabled
INDEX_NUMBER	FIXED BIN(31)	Current index number
INDEXED	BIT(1)	File is/is not an indexed sequential file
INITIAL_FILL	BIT(1)	INITIAL_FILL option is enabled/disabled
MAXIMUM_RECORD_NUMBER	FIXED BIN(31)	Relative file maximum relative record
MAXIMUM_RECORD_SIZE	FIXED BIN(31)	Largest record size
MULTIBLOCK_COUNT	FIXED BIN(31)	Multiblock count (disk files only)
MULTIBUFFER_COUNT	FIXED BIN(31)	Multibuffer count
NO_SHARE	BIT(1)	NO_SHARE option is enabled/disabled
OWNER_GROUP	FIXED BIN(31)	Group number of file's owner
OWNER_MEMBER	FIXED BIN(31)	Member number of file's owner
OWNER_PROTECTION	CHAR(4) VARYING	Protection for file's owner
RETRIEVAL_POINTERS	FIXED BIN(31)	Number of mapping pointers
PRINTER_FORMAT	BIT(1)	Records have printer carriage control
READ_AHEAD	BIT(1)	READ_AHEAD option is enabled/disabled
READ_CHECK	BIT(1)	READ_CHECK option is enabled/disabled

Table 16-6 (Cont.): ENVIRONMENT Option Values Returned by DISPLAY

Member Name	Data Type of Value Returned	Meaning
RECORD_ID_ACCESS	BIT(1)	File is opened for access by record identification
REWIND_ON_CLOSE	BIT(1)	REWIND_ON_CLOSE option is enabled/disabled
REWIND_ON_OPEN	BIT(1)	REWIND_ON_OPEN option is enabled/disabled
SCALARVARYING	BIT(1)	SCALARVARYING option is enabled/disabled
SHARED_READ	BIT(1)	SHARED_READ option is enabled/disabled
SHARED_WRITE	BIT(1)	SHARED_WRITE option is enabled/disabled
SPOOL	BIT(1)	SPOOL option is enabled/disabled
SUPERSEDE	BIT(1)	SUPERSEDE option is enabled/disabled
SYSTEM_PROTECTION	CHAR(4) VARYING	Protection for system users
TEMPORARY	BIT(1)	TEMPORARY option is enabled/disabled
TRUNCATE	BIT(1)	TRUNCATE option is enabled/disabled
WORLD_PROTECTION	CHAR(4) VARYING	Protection for world users
WRITE_BEHIND	BIT(1)	WRITE_BEHIND option is enabled/disabled
WRITE_CHECK	BIT(1)	WRITE_CHECK option is enabled/disabled

Table 16–7 summarizes the file attribute information returned by DISPLAY, including

- PL/I file description attributes and options specified for the file.
- The file's organization, expanded file specification, and, for an indexed sequential file, the number of keys it has.

All names in Table 16–7 are second-level members of the structure PLI_FILE_DISPLAY.

Table 16–8 lists the names of the structure members that contain information about the device to which a file is written or from which the file is to be read. All of the names in Table 16–8 are third-level members of the structure PLI_FILE_DISPLAY; they each appear within the following minor structures, which have identical declarations:

- DEVICE
- SPOOLING_DEVICE

If the field PLI_FILE_DISPLAY.DEVICE.SPL is true, then the members of the minor structure DEVICE contain information about the device that is spooled, and the members of the minor structure PLI_FILE_DISPLAY.SPOOLING_DEVICE contain information about the intermediate, or spooling, device.

All fields within these structures are BIT(1) values.

16.4.2 EXTEND Built-In Subroutine

The EXTEND built-in subroutine increases the amount of space allocated to a disk file. Its calling sequence is

```
CALL EXTEND (file-reference,integer-expression) ;
```

file-reference

Specifies the name of a file variable or constant associated with the file to be extended. If the file is not currently open, the EXTEND subroutine opens it with the OUTPUT attribute in order to extend it.

integer-expression

Is a fixed binary expression in the range 0 to 4,294,967,295, specifying the number of 512-byte disk blocks to be added to the file. If 0 is specified, PL/I uses the default extension quantity for the file.

To specify a value larger than 2,147,483,647 (the largest value that can be contained in a fixed binary integer in PL/I), you must express the number as a negative value; RMS interprets the number as an unsigned integer.

Use the EXTEND built-in subroutine to explicitly extend a file during processing. Normally, whenever an output operation causes a file to exceed its allocated space, RMS extends it automatically, using the current extension size value. The default value that RMS uses to extend a file is set by the ENVIRONMENT option EXTENSION_SIZE.

Table 16–7: File Attribute Information Returned by DISPLAY

Member Name	Data Type of Value Returned	Meaning
COLUMN_NUMBER	FIXED BIN(31)	Current column (stream output files only)
DIRECT	BIT(1)	File has/does not have DIRECT attribute
EXPANDED_TITLE	CHAR(128) VARYING	Expanded file specification
FILE_ORGANIZATION	CHAR(3)	SEQ, REL, or IDX
FORTRAN_FORMAT	BIT(1)	File has/does not have FTN (ASA) carriage control
INPUT	BIT(1)	File has/does not have INPUT attribute
KEYED	BIT(1)	File has/does not have KEYED attribute
LINE_NUMBER	FIXED BIN(31)	Current line number (stream output files only)
LINESIZE	FIXED BIN(31)	File's line size (stream output files only)
NUMBER_OF_KEYS	FIXED BIN(31)	Number of keys (indexed sequential files only)
OUTPUT	BIT(1)	File has/does not have OUTPUT attribute
PAGE_NUMBER	FIXED BIN(31)	Current page number (PRINT files only)
PAGESIZE	FIXED BIN(31)	Page size (PRINT files only)
PRINT	BIT(1)	File has/does not have PRINT attribute
RECORD	BIT(1)	File has/does not have RECORD attribute
SEQUENTIAL	BIT(1)	File has/does not have SEQUENTIAL attribute
STREAM	BIT(1)	File has/does not have STREAM attribute
UPDATE	BIT(1)	File has/does not have UPDATE attribute

Table 16-8: Device Information Returned by DISPLAY

Member Name	Meaning
ALL	Device is/is not allocated.
AVL	Device is/is not online and available.
CCL	Device has carriage control.
DIR	Device is/is not directory structured.
DMT	Device is/is not marked for dismounting.
ELG	Device is/is not enabled for error logging.
FOD	Device is/is not file-oriented.
FOR	Device is/is not a foreign device.
GEN	Device is/is not a generic device.
IDV	Device is/is not capable of input.
MBX	Device is/is not a mailbox.
MNT	Device is/is not mounted.
NET	Device is/is not a network device.
ODV	Device is/is not capable of output.
RCK	Device performs read checking.
REC	Device is/is not a record-oriented device (terminal or line printer, for example).
RND	Device is/is not random-access device.
RTM	Device is/is not a real-time device.
SDI	Device has a master directory only.
SHR	Device is/is not shareable.
SPL	Device is/is not spooled.
SQD	Device is/is not sequential block-oriented (magnetic tape).
SWL	Device is/is not currently software write-locked.
TRM	Device is/is not a terminal.
WCK	Device performs write checking.

You can improve the performance of a program that is going to add a large number of records to a file by an explicit call to EXTEND before records are added. RMS does not then need to extend the file during the actual I/O operations.

16.4.3 FLUSH Built-In Subroutine

The FLUSH built-in subroutine writes all RMS buffers that have been modified and preserves all the attributes of the file. This subroutine provides the ability to checkpoint a file during its processing and ensure its integrity. Its calling sequence is

```
CALL FLUSH (file-reference);
```

file-reference

Specifies the name of the file variable or file constant associated with the file whose buffers are to be flushed. If the file is not currently open, the FLUSH subroutine performs no operation.

Use the FLUSH subroutine to explicitly request RMS to write all internal file buffers back to the file. This subroutine is called implicitly by the REWIND and NEXT_VOLUME built-in subroutines.

16.4.4 NEXT_VOLUME Built-In Subroutine

The NEXT_VOLUME built-in subroutine performs the positioning and labeling functions necessary when the next volume is required during I/O to a magnetic tape file that spans more than one physical tape volume. Its calling sequence is

```
CALL NEXT_VOLUME (file-reference);
```

file-reference

Specifies the name of the file constant or file variable associated with the tape volume set being processed. If the file is not currently open, the NEXT_VOLUME subroutine implicitly opens it with the attributes specified in the file's declaration.

When a multivolume tape file is being read or written, volume switching is normally transparent to the PL/I program. RMS, and the magnetic tape Ancillary Control Program (ACP), perform all the steps necessary to ensure that the next required volume is physically mounted, initialized, and verified.

However, when a program wishes to advance to the next volume before reaching the end of the current volume on input, or before the end of the tape is reached on output, it can call the NEXT_VOLUME built-in subroutine. This subroutine performs all the necessary volume checking when a multivolume tape file is being read. When a file is being written, the subroutine writes the appropriate information on the output tapes.

16.4.5 REWIND Built-In Subroutine

The REWIND built-in subroutine positions a file so that the next record to be read will be the first record in the file or index. Its calling sequence is

```
CALL REWIND (file-reference);
```

file-reference

Specifies the name of the file constant or file variable associated with the file to be rewound. If the file is not currently open, the REWIND subroutine implicitly opens it with the attributes specified in the file's declaration.

Use this subroutine to begin processing a file at its logical beginning. This subroutine is valid for disk files of all organizations and for sequential files on tape volumes. The position of the file following the call to the REWIND subroutine is as follows:

- For a sequential file, the REWIND subroutine positions the file at its first record.
- For a relative file, the REWIND subroutine positions the file at its first occupied cell.
- For an indexed sequential file, the REWIND subroutine positions the file at the lowest key value in the current index.
- If a magnetic tape file is on a single volume, the volume is rewound. If the tape file exists on a multivolume tape set, the REWIND subroutine rewinds the file to the beginning of the volume set.

16.4.6 SPACEBLOCK Built-In Subroutine

The SPACEBLOCK built-in subroutine positions a file forward or backward a specified number of blocks. This subroutine can be used to process unlabeled magnetic tapes, as well as sequential disk files being processed with block I/O. Its calling sequence is

```
CALL SPACEBLOCK (file-reference,integer-expression);
```

file-reference

Specifies the name of the file constant or file variable to be spaced. If the file is open, it must have been opened with the BLOCK_IO option. If the file is not open, the SPACEBLOCK subroutine opens it with the BLOCK_IO option.

integer-expression

Is a fixed binary expression specifying the number of blocks to be spaced forward or backward. If the expression is negative, the file is spaced backward the specified number of blocks; if positive, forward the specified number of blocks.

16.5 File Error Handling

VAX-11 PL/I uses the standard PL/I ON condition names to signal run-time errors that occur during file operations. The ON conditions that are signaled, and the circumstances under which they are signaled, are as follows:

- The UNDEFINEDFILE condition is signaled whenever a file cannot be opened.
- The ENDFILE condition is signaled when the end-of-file is reached during an input operation.
- The ENDPAGE condition is signaled for a file with the PRINT attribute when the current line number exceeds the page size specified for the file.
- The KEY condition is signaled for a file with the KEYED attribute when any error occurs involving the interpretation, writing, or specification of a key.
- The ERROR condition is signaled for all other file-related errors.

To handle any of those conditions in a PL/I procedure, you can establish an ON-unit to receive control if the specified condition is signaled. For example:

```
ON UNDEFINEDFILE (INFILE) OPEN FILE (INFILE)
              TITLE ('SYS$INPUT');
```

The ON statement provides a default title for the file INFILE.

16.5.1 Values Returned by PL/I Built-In Functions

An ON-unit can be a generalized error-handling routine, written so that it responds to specific errors or so that it prints an error message. The PL/I built-in functions that provide meaningful information in an ON-unit to handle a file system error are

- ONCODE
- ONFILE
- ONKEY

Whenever an error is signaled, the built-in function ONCODE makes available the condition value associated with the specific error. When a PL/I program is executing under control of the VAX/VMS operating system, the value returned by the ONCODE built-in function is a unique 32-bit condition value from the system, from RMS, or from the PL/I run-time system, which indicates the reason for the error. Section 16.5.2 contains an example of an ON-unit that examines the value returned by ONCODE.

The built-in function ONFILE returns a character string giving the name of the file constant on which the error occurred.

If the file was being accessed by key, the ONKEY built-in function returns the key value that caused the error to be signaled.

16.5.2 Writing an Error Handler

You can write an ON-unit to detect and correct errors that occur during file operations. The example below illustrates an ON-unit that detects whether a record with a given key value was not found or whether an attempt was made to write a record whose key duplicates the value of an existing key.

```
ON KEY(STATE_FILE) BEGIN;
DECLARE (RMS$_RNF, RMS$_DUP) GLOBALREF
        FIXED BINARY(31) VALUE;

/* Check for a record not found */
     IF ONCODE() = RMS$_RNF THEN DO; /* if record not found */

        PUT SKIP EDIT(STATENAME,'Not found.')
                 (A,X,A);
        STOP;
        END;

/* Check for duplicate key */
     ELSE IF ONCODE = RMS$_DUP THEN DO;
        PUT SKIP EDIT('Record already exists for',
                       STATENAME)
                 (A,X,A);
        STOP;
        END;
   END;
```

In this example, the ON-unit declares symbolic names for two specific status values returned by ONCODE:

- The value RMS$_RNF indicates that no record exists with the specified key value.
- The value RMS$_DUP indicates that a record already exists with the specified key in an index for which duplicate keys are not allowed.

In an ON-unit for the KEY condition, ONCODE may also return the value associated with the status code RMS$_KEY. This code indicates that a key value is invalid; for example, it is an incorrect data type.

The symbolic names for RMS status codes must be declared with the GLOBALREF and VALUE attributes because the names are defined as global symbols by the VAX/VMS system.

16.5.3 Default Error Handling

If a file system error occurs during the execution of a PL/I statement, the PL/I run-time system signals either the specific PL/I condition name or the ERROR condition. If no user-specified ON-units exist to handle either the specific PL/I condition or the ERROR condition, PL/I performs its default condition handling.

If any active procedure specified OPTIONS (MAIN), a default PL/I condition handler is present and executed. It prints a PL/I run-time error message. If there is no default PL/I handler, the error signal is passed to the default condition handler established by VAX/VMS, which prints the message associated with the RMS error. If the error was a fatal error, the handler terminates the program; otherwise, the program continues.

The following example illustrates the type of message that the PL/I run-time system displays when an error occurs during an I/O operation:

```
%PLI-F-ERROR, PL/I ERROR condition.
-PLI-I-IOERROR, I/O error on file 'STATE_FILE'
-PLI-I-FILENAME, File name:
                  '_DB7:[PROJECT]STDATA.DAT;'
-PLI-I-NOTKEYD, Not a KEYED file.
%TRACE-F-TRACEBACK, symbolic stack dump follows

module          routine
name            name       line   relative PC  absolute PC

FLOWERS         BEGIN%35   35       00000085      00000C88
FLOWERS         BEGIN%29   29       000000BD      00000C02
FLOWERS         FLOWERS    25       000000D3      00000B42
```

In this example, the error occurred because a keyed I/O statement was specified for a file lacking the KEYED attribute. For an explanation of the information in a traceback message, see Section 5.1.2.

Chapter 17

Stream Input/Output

Stream I/O is one of the two general kinds of I/O performed by PL/I (record I/O, described in Chapter 18, is the other). In stream I/O, more than one record or line can be processed by a single statement, and, conversely, multiple statements can process a single line or record. In record I/O, only one record of a file is processed by each READ or WRITE statement.

Stream input and output are performed by the statements GET and PUT, respectively. Both statements can perform either list-directed or edit-directed operations.

This chapter describes

- Statements used for stream I/O—GET, PUT, and FORMAT.
- Processing and positioning that take place during stream I/O operations.
- Format items and methods of combining them into format specifications for edit-directed stream I/O operations.

The rest of this section describes the attributes and options applicable to stream I/O. You can use PRINT and STREAM when you declare a file or when you open it. LINESIZE and PAGESIZE are only valid when you open the file.

LINESIZE Option

The LINESIZE option specifies the maximum number of characters that can be output in a single line when the PUT statement writes data to a file with the STREAM and OUTPUT attributes. Its format is

```
LINESIZE(expression)
```

expression

A fixed-point binary expression in the range 1 to 32767, giving the number of characters per line.

The value specified in the LINESIZE option is used as the output line length for all subsequent output operations on the stream file, overriding the system default line size.

The default line size is as follows:

- If the output is to a physical record-oriented device, such as a line printer or terminal, the default line size is the width of the device.
- If the output is to a print file, the default line size is 132.
- If the output is to a nonrecord device (magnetic tape), the default line size is 510.

The line size is used by output operations to determine whether output will be placed on the current line or on the next line.

PAGESIZE Option

The PAGESIZE option is used in the OPEN statement to specify the maximum number of lines that can be written to a print file without signaling the ENDPAGE condition. The PAGESIZE option is valid only for print files. Its format is

```
PAGESIZE(expression)
```

expression

A fixed-point binary expression in the range 1 to 32767, giving the number of lines per page. The maximum value for a print file's page number is 32767. If a program generates a value in excess of 32767, a run-time error occurs.

The value specified in the PAGESIZE option is used as the output page length for all subsequent output operations on the print file, overriding the system default page size (see Section 17.2.2). During output operations, the ENDPAGE condition is signaled the first time that the specified page size is exceeded.

PRINT Attribute

The PRINT attribute is used to declare a print file. The file SYSPRINT, used as the default output by PUT statements, is also a print file.

Print files are stream output files with special formatting characteristics (see Section 17.2.2). The PRINT attribute implies the OUTPUT and STREAM attributes, and conflicts with the INPUT, RECORD, UPDATE, KEYED, SEQUENTIAL, and DIRECT attributes.

STREAM Attribute

The STREAM file description attribute indicates that the file consists of ASCII characters and that it will be processed using GET and PUT statements.

The STREAM attribute is implied by the PRINT attribute. It is also supplied by default for a file that is implicitly opened with a GET or PUT statement.

Specify the STREAM attribute in a DECLARE statement for a file identifier or in the OPEN statement that opens the file.

The STREAM attribute directly conflicts with the RECORD, KEYED, DIRECT, SEQUENTIAL, and UPDATE attributes.

17.1 Statements for Stream I/O

The three sections that follow describe the GET, PUT, and FORMAT statements.

17.1.1 GET Statement

The GET statement acquires data from an input stream, which is either a stream file or a character-string expression. The input file may be a file declared with the STREAM attribute, or it may be the default file SYSIN, which is commonly associated with the user's default input device.

The GET statement has several forms. They are summarized in Figure 17-1 and described below.

```
GET  EDIT (input-target★,...) (format-specification,...)
  ┌ FILE(file-reference)★               ┐
  │       [SKIP[(expression)]]★         │
  │        [OPTIONS(option,...)]★       │
  │                                     │
  └ STRING(expression)★                 ┘
     ;

GET  LIST (input-target★,...)
  ┌ FILE(file-reference)★               ┐
  │       [SKIP[(expression)]]★         │
  │        [OPTIONS(option,...)]★       │
  │                                     │
  └ STRING(expression)★                 ┘
     ;

GET  [FILE(file-reference)]★ SKIP [(expression)] ;

Options★
NO_ECHO
NO_FILTER
PROMPT(expression)
PURGE_TYPE_AHEAD

★Syntax elements common to two or more forms

                                          ZK-031-81
```

Figure 17-1: Forms of the GET Statement

17.1.1.1 Common Syntax Elements

The syntax elements flagged with a star in Figure 17–1 are common to two or more forms of the GET statement. This section describes those elements. The sections that follow describe aspects of GET EDIT, GET LIST, and GET SKIP that are unique to each form.

input-target

The names of one or more variables to be assigned values from the input stream. The input targets must be separated by commas. An input target has the following forms:

1. Reference

   where the reference is to a scalar or aggregate variable of any computational type. If the reference is to an array, data is assigned to array elements in row-major order. If the reference is to a structure, data is assigned to structure members in the order of their declaration.

2. (input-target,... DO reference=expression[TO expression] [BY expression][WHILE(expression)[UNTIL(expression)])

   where the input target may be of any of these forms, and the references and expressions are as for the DO statement. Notice that the parentheses surrounding this form of input target are in addition to those surrounding the entire input list.

3. (input-target,...DO reference=expression [REPEAT expression] [WHILE (expression)[UNTIL (expression)])

   where the input target may be of any of these forms, and the references and expressions are as for the DO statement. Notice that the parentheses surrounding this form of input target are in addition to those surrounding the entire input list.

FILE(file-reference)

An option specifying that the input stream is a file; the reference is to a declared file variable or constant. If neither the FILE option nor the STRING option is specified, PL/I assumes the file SYSIN. This file is associated with the default system input file SYS$INPUT, which is usually a terminal.

If a file is specified but not currently open, PL/I opens it with the attributes STREAM and INPUT. The UNDEFINEDFILE condition is signaled if the file cannot be opened.

STRING(expression)

An option specifying that the input stream is a character-string ex-

pression. The STRING option cannot be used with the FILE option, nor can it be used with the OPTIONS or SKIP option.

SKIP [(expression)]

An option that advances the input file a specified number of lines before processing the input list. It may be used only with the implied or explicit FILE option. The expression, if specified, indicates the number of lines to be advanced; if it is omitted, the default is to skip to the next line. The SKIP option is always executed first, before any other input or positioning of the input file, and regardless of its position in the statement.

OPTIONS (option,...)

An option that specifies one or more of the following options. It may be used only with the default or explicit FILE option; it cannot be used with the STRING option. The options must be separated by commas and enclosed in parentheses.

NO_ECHO

Specifies, when the input device is a terminal, that the data entered at the terminal will not be displayed as it is entered. This option is ignored for other input devices.

NO_FILTER

Specifies, when the input device is a terminal, that the recognition of CTRL/U, CTRL/R, and the DEL key is to be suppressed. These characters are interpreted as terminators. This option is ignored for other input devices.

PROMPT (string-expression)

Specifies, when the input device is a terminal, a character-string prompt to be displayed prior to actual input. The string expression can be 1 to 254 characters long.

PURGE_TYPE_AHEAD

Specifies, when the input device is a terminal, that all data in the terminal's type-ahead buffer be deleted before the input operation. This option is ignored for other input devices.

17.1.1.2 GET EDIT

The GET EDIT statement acquires fields of character-string data from an input stream (a stream file or a character-string expression). The stream file may be a declared file or the default file SYSIN. GET EDIT converts the character strings under control of a format specification and assigns the resulting values to a specified list of input targets (variables). It also allows input of characters from selected positions in the input stream.

The form of the GET EDIT statement is

```
GET EDIT (input-target,...) (format-specification,...)

    ┌FILE(file-reference)          ┐
    │       [SKIP[(expression)]]   │
    │        [OPTIONS(option,...)] │
    │                              │
    └STRING(expression)            ┘
      ;
```

input-target

The names of one or more variables to be assigned values from the input stream; described fully in Section 17.1.1.1. For a discussion of the matching of format items to input targets, see Section 17.3.

format-specification

A list of format items to control the conversion of data items in the input list. Section 17.3 describes format items and specifications.

FILE(file-reference)
STRING(expression)
SKIP [(expression)]
OPTIONS (option,...)

These common syntax elements are described in Section 17.1.1.1.

The following examples demonstrate GET EDIT.

```
GET EDIT (FIRST,MID_INITIAL,LAST)
      (A(12),A(1),A(20));
```

This statement reads the next 33 characters from the default stream input file (SYSIN) and assigns them to FIRST (12 characters), MID_INITIAL (1 character), and LAST (20 characters).

```
GET EDIT (SOCIAL_SECURITY) (A(12))
      FILE (SOCIAL) SKIP (12) ;
```

This statement opens (if necessary) the stream file SOCIAL, advances 12 lines, reads the first 12 characters of the line, and assigns the characters to the variable SOCIAL_SECURITY.

```
GET EDIT (N, (A(I) DO I=1 TO N))
      (F(4),SKIP,100 F(10,5));
```

The dimension of A is less than or equal to 100. The value of N is read from the input stream using the format item F(4). The process then skips to the next line (record), and N elements are read into the array A. Each element is read using the format item F(10,5).

```
GET EDIT (NAME.FIRST,NAME.LAST)
      (A(10),X(3),A(20))
      STRING('Philip    A. Rothberg              ');
```

This statement assigns 'Philip ' to the structure member NAME.FIRST, skips the middle initial, period, and space, and assigns 'Rothberg ' to NAME.LAST.

For more examples, see Section 17.3.

17.1.1.3 GET LIST

The GET LIST statement acquires character-string or bit-string data from an input stream (a stream file or a character-string expression). The stream file may be a declared file or the default file SYSIN. The acquired character strings and bit strings are assigned to input targets named in the GET LIST statement; the strings are converted automatically to the targets' data types.

Use the GET LIST statement to read "unformatted" data from a stream file or character string. Because you need not place the input data in specific columns, GET LIST is useful for acquiring data from a terminal.

The form of the GET LIST statement is

```
GET LIST (input-target,...)

    ┌FILE(file-reference)          ┐
    │       [SKIP[(expression)]]   │
    │        [OPTIONS(option,...)] │
    │                              │
    └STRING(expression)            ┘
      ;
```

input-target

The names of one or more variables to be assigned values from the input stream; described fully in Section 17.1.1.1.

FILE(file-reference)
STRING(expression)
SKIP [(expression)]
OPTIONS (option,...)

Each of these elements is described in Section 17.1.1.1.

The items to be read into the input targets by GET LIST are separated by a space or a single comma. Multiple spaces are treated as a single space, and a comma may be surrounded by spaces. The following rules apply:

- GET LIST treats an acquired item as a character string unless it has the form of a bit string, in which case GET LIST treats it as a bit string. For example, the input item '7' is a 1-character string; the item '7'B3 is the 3-bit string '111'B.
- No items can be split across lines unless the split occurs inside a quoted string or ENVIRONMENT (IGNORE_LINE_MARKS) is

specified. In these cases, the acquired item does not include a carriage return or end-of-line.

- Character strings need not be enclosed in apostrophes unless they contain a space or comma or are written on more than one line. When apostrophes do enclose a character string, an apostrophe within the string is written as two apostrophes; for instance, to input the word *isn't*, write *'isn''t'*.

- When a line begins with a comma or when two commas appear in the line without intervening nonspace characters, the item in the input-target list corresponding to that item is not updated. The target retains whatever value it contained before GET LIST was executed.

- Every input field, except the last one in a line, must be terminated by a space or comma. Unless it occurs in a quoted string (or ENVIRONMENT (IGNORE_LINE_MARKS) is used), a carriage return or the end of a line in a file acts as a field terminator. The value thus acquired does not have a space or comma at the end.

- Input fields are also terminated by the end-of-file (FILE option) or end-of-string (STRING option) unless the end is encountered inside a quoted string.

- If an input request from GET LIST encounters a null record, the corresponding input target is nulled (if a string) or assigned a value of zero (if arithmetic). A null input record means a null record in a file or, if the input is from a terminal, a carriage return with no other input. If ENVIRONMENT (IGNORE_LINE_MARKS) is used for the input file, record terminators such as the carriage return are ignored, and the GET LIST statement waits until the input request is satisfied.

- The ERROR condition is signaled whenever a data item in the stream cannot be converted to the data type of the corresponding item in the input-target list.

- The ENDFILE condition is signaled if the end of the file is encountered during file input. The ERROR condition is signaled if the expression in the STRING option does not contain enough characters to complete processing of the input-target list.

The following examples demonstrate GET LIST.

```
GETS: PROCEDURE OPTIONS(MAIN);

DECLARE NAME CHARACTER(80) VARYING;
DECLARE AGE FIXED;
DECLARE (WEIGHT,HEIGHT) FIXED DECIMAL(5,2);
DECLARE SALARY PICTURE '$$$$$$V.$$';
DECLARE DOSAGE FLOAT;
```

```
DECLARE INFILE STREAM INPUT FILE;
DECLARE OUTFILE PRINT FILE;

GET FILE(INFILE)
     LIST(NAME,AGE,WEIGHT,HEIGHT,SALARY,DOSAGE);

PUT FILE(OUTFILE)
     LIST(NAME,AGE,WEIGHT,HEIGHT,SALARY,DOSAGE);
END GETS;
```

If the file INFILE.DAT contains the data

```
'Thomas R. Dooley',33,150.60,5.87,15000.50,5E-6,
```

then the program GETS writes the following output to OUTFILE.DAT:

```
Thomas R. Dooley 33 150.60  5.87  $15000.50  4.9999999E-06
```

In the input file (INFILE.DAT) the string 'Thomas R. Dooley' was surrounded by apostrophes so that the spaces between words would not be interpreted as field separators.

```
GSTR: PROCEDURE OPTIONS(MAIN);

DECLARE STREXP CHARACTER(80) VARYING;
DECLARE (A,B,C,D,E) FIXED;
DECLARE OUTFILE STREAM OUTPUT FILE;

OPEN FILE(OUTFILE) TITLE('GSTR.OUT');

STREXP = '1,2,3,4,5';
GET STRING(STREXP) LIST(A,B,C,D,E);
PUT FILE(OUTFILE) LIST(A,B,C,D,E);

END GSTR;
```

The program GSTR writes the following output to GSTR.OUT:

```
     1       2       3       4       5
```

17.1.1.4 GET SKIP

The GET SKIP statement positions the input file at the start of a new line. This form of the GET statement is

GET [FILE(file-reference)] SKIP [(expression)] ;

file-reference

The name of the file to be advanced one or more lines. The defaults for the file reference are specified in Section 17.1.1.1.

expression

An integer expression giving the number of lines to be advanced. The default is one line.

17.1.2 PUT Statement

The PUT statement transfers data from the program to the output stream, which may be either a stream file or a character-string variable. The output file may be a declared file or the default file SYSPRINT.

The PUT statement has several forms. These forms are summarized in Figure 17–2 and described individually below.

17.1.2.1 Common Syntax Elements

The syntax elements flagged with a star in Figure 17–2 are common to two or more forms of the PUT statement. This section describes those elements. The sections that follow describe aspects of PUT EDIT, PUT LINE, PUT LIST, PUT PAGE, and PUT SKIP that are unique to each.

output-source

A construct that specifies one or more expressions to be placed in the output stream. The output sources must be separated by commas. An output source has the following forms:

1. expression

   where the expression is of any computational type, including a reference to a scalar or aggregate variable. If the reference is to an array, data is output from array elements in row-major order. If it is to a structure, data is output from structure members in the order of their declaration.

2. (output-source... DO reference=expression[TO expression] [BY expression][WHILE(expression)][UNTIL(expression)])

   where the output source may be of any of these forms, and the references and expressions are as for the DO statement. Notice that the parentheses surrounding this form of output source are in addition to the parentheses surrounding the entire output-source list.

3. (output-source,... DO reference=expression [REPEAT expression][WHILE(expression)][UNTIL(expression)])

   where the output source may be of any of these forms, and the references and expressions are as for the DO statement. Notice that the parentheses surrounding this form of output source are in addition to the parentheses surrounding the entire output-source list.

```
PUT  EDIT (output-source★,...) (format-specification,...)
  ┌ FILE(file-reference)★                              ┐
  │      [PAGE]★ [LINE(expression)]★                   │
  │       [SKIP[(expression)]]★                        │
  │        [OPTIONS(option)]★                          │
  │                                                    │
  └ STRING(reference)★                                 ┘
      ;

PUT [ FILE (file-reference)★ ] LINE (expression) ;

PUT  LIST (output-source,...)★
  ┌ FILE(file-reference)★                              ┐
  │      [PAGE]★ [LINE(expression)]★                   │
  │       [SKIP[(expression)]]★                        │
  │        [OPTIONS(option)]★                          │
  │                                                    │
  └ STRING(reference)★                                 ┘
      ;

PUT [ FILE(file-reference)★ ] PAGE;

PUT [ FILE(file-reference)★ ] SKIP [(expression)] ;

Option★
CANCEL_CONTROL_O

★Syntax elements common to two or more forms
```

ZK-032-81

Figure 17-2: Forms of the PUT Statement

FILE(file-reference)

An option specifying that the output stream is a stream file; the reference is to a declared file variable or constant. If neither the FILE option nor the STRING option is specified, PL/I uses the default file SYSPRINT; this print file is associated with the default system output file SYS$OUTPUT, which in turn is generally associated with the user's terminal.

If a specified file is not currently open, PL/I opens it with the attributes STREAM and OUTPUT.

PAGE

An option that advances the output file to a new page before any data is transmitted. The PAGE option may be used only with implied or explicit print files. The file is positioned at the beginning of the next page, and the current page number is incremented by 1. The PAGE, LINE, and SKIP options are always executed, in that order, before

any other output or file-positioning operations. The page size is either the default value or the specific value that you have established for the file.

LINE (expression)

An option that advances the output file to a specified line. The LINE option may be used only with implied or explicit print files. The expression must yield an integer i. Blank lines are inserted in the output file so that the next output data appears on the *i*th line of a page.

If the file is currently positioned at the beginning of line i, no operation is performed by the LINE option.

If the file is currently positioned before line i, and i is less than or equal to the page size, then blank lines are inserted following the current line until line i is reached.

If the file is currently positioned at or beyond line i but not at the beginning of line i, then the remainder of the page (the portion between the current line and the current page size) is filled with blank lines. The ENDPAGE condition is signaled.

When the LINE option is used within an ENDPAGE ON-unit, it causes a skip to the next page.

SKIP [(expression)]

An option that advances a specified number of lines from the current line. The SKIP option may be used only with the implied or explicit FILE option. The expression must yield an integer i, which must not be negative and must be greater than 0 except for print files. If the expression is omitted, i equals 1.

If the file is not a print file, i–1 blank lines are inserted following the current line, and subsequent output of data begins at the beginning of (current line)+i.

If the file is a print file, i=0 causes a return to the beginning of the current line. If i is greater than 0, and either the current line exceeds the page size or the page size is greater than or equal to the current line plus i, then i–1 blank lines are inserted. Otherwise, the remainder of the current page is filled with blank lines, and the ENDPAGE condition is signaled.

On output devices with the space-suppression feature, SKIP(0) can be used to cause overprinting, underscoring, and so forth.

OPTIONS (CANCEL_CONTROL_O)

A statement option that may be included only with the implied or explicit FILE option. It specifies, when the output device is a termi-

nal, that the effect of CTRL/O is disabled prior to output, thus ensuring that the beginning of the output list is displayed. Use this option on a PUT statement that you want to display regardless of whether previous output has been interrupted by CTRL/O. This option is ignored when the output device is not a terminal.

STRING(reference)

An option specifying that the output stream is the referenced character-string variable. The STRING option cannot be used in the same statement with FILE, OPTIONS, PAGE, LINE, or SKIP.

17.1.2.2 PUT EDIT

The PUT EDIT statement takes output sources (variables and expressions) from the program, converts the results to characters under control of a format specification, and places the resulting character strings in the output stream (either a stream file or a character-string variable).

With PUT EDIT, the format of the output data is controlled by the program.

The form of the PUT EDIT statement is

```
PUT EDIT (output-source,...) (format-specification,...)

      ┌FILE(file-reference)             ┐
      │     [PAGE] [LINE(expression)]   │
      │      [SKIP[(expression)]]       │
      │       [OPTIONS(option,...)]     │
      │                                 │
      └STRING(reference)                ┘

      ;
```

output-source

A construct that specifies one or more expressions to be placed in the output stream. Section 17.1.2.1 contains a complete description of this element. For a discussion of the matching of format items to output sources, see Section 17.3.

format-specification

A list of format items to control the conversion of data items in the output list. Section 17.3 describes format items and specifications.

FILE(file-reference)
PAGE
LINE (expression)
SKIP [(expression)]

OPTIONS (CANCEL_CONTROL_O)
STRING(reference)

Section 17.1.2.1 contains descriptions of these elements.

The following example demonstrates the PUT EDIT statement.

```
PUTE: PROCEDURE OPTIONS(MAIN);

DECLARE SOURCE FIXED DECIMAL(7,2);

DECLARE OUTFILE PRINT FILE;

OPEN FILE(OUTFILE) TITLE('PUTE.OUT');

SOURCE = 12345.67;

PUT SKIP FILE(OUTFILE) EDIT(SOURCE) (F(8,2));
PUT SKIP FILE(OUTFILE) EDIT(SOURCE) (E(13));
PUT SKIP FILE(OUTFILE) EDIT(SOURCE) (A);
PUT SKIP FILE(OUTFILE) EDIT('American: ',SOURCE)
        (A,P'ZZ,ZZZV.ZZ');
PUT SKIP FILE(OUTFILE) EDIT('European: ',SOURCE)
        (A,P'ZZ.ZZZV,ZZ');

END PUTE;
```

The program PUTE writes the following output to PUTE.OUT:

```
12345.67
 1.234567E+04
  12345.67
American: 12,345.67
European: 12.345,67
```

17.1.2.3 PUT LINE

The PUT LINE statement advances a print file to a specified line. The form of the PUT LINE statement is

PUT [FILE (file-reference)] LINE (expression);

file-reference

A reference to the file to which the statement applies. The file must be a print file. The defaults for the file reference are specified in Section 17.1.2.1.

expression

An expression giving a line in the print file, relative to the top of the current page. The expression must yield an integer. The processing performed by PUT LINE for various values of the expression is described in Section 17.1.2.1.

17.1.2.4 PUT LIST

The PUT LIST statement specifies a list of output sources (variables and expressions) whose results are converted to character strings and transmitted to the output stream. If the output file is a print file, the output character strings are separated by enough spaces to start the new string at a tab boundary. Otherwise, the strings are separated by single spaces. For a nonprint file, character-string data is output with enclosing apostrophes; for a print file, there are no apostrophes. For both print and nonprint files, bit-string data is output with enclosing apostrophes, followed by the letter B.

With PUT LIST, the conversion of the output sources and formatting of the output data are automatic.

The form of the PUT LIST statement is

```
PUT LIST (output-source,...)

    ┌FILE(file-reference)              ┐
    │     [PAGE] [LINE(expression)]    │
    │      [SKIP[(expression)]]        │
    │       [OPTIONS(option,...)]      │
    │                                  │
    └STRING(reference)                 ┘
       ;
```

output-source

A construct that specifies one or more expressions to be placed in the output stream; see Section 17.1.2.1.

FILE(file-reference)
PAGE
LINE (expression)
SKIP [(expression)] OPTIONS (CANCEL_CONTROL_O)
STRING(reference)

Section 17.1.2.1 fully describes these elements.

The following example demonstrates the PUT LIST statement.

```
PUTL: PROCEDURE OPTIONS(MAIN);

DECLARE I FIXED BINARY,
        F FLOAT,
        P PICTURE '99V.99',
        S CHAR(10);

DECLARE INFILE STREAM INPUT FILE;
DECLARE OUTFILE PRINT FILE;
```

```
OPEN FILE(INFILE) TITLE('PUTL.IN');
OPEN FILE(OUTFILE) TITLE('PUTL.OUT');

GET FILE(INFILE) LIST (I,F,P,S);
PUT FILE(OUTFILE) SKIP LIST (I,F,P,S);

END PUTL;
```

If the file PUTL.IN contains the data

```
2,3.54,22.33,'A string'
```

then the program PUTL writes the following output to PUTL.OUT:

```
                2    3.5400000E+00   22.33    A string
```

For print files, each output item is written at the next tab position. Floating-point values are represented in floating-point notation, and character values are not enclosed in apostrophes.

If the file PUTL.OUT is changed from a print file to a nonprint file (by substituting OUTPUT for PRINT in the declaration of OUTFILE), the output of the program becomes

```
                2  3.5400000E+00 22.33 'A string  '
```

Note that PUT LIST now separates items with a space and encloses the character-string item in apostrophes. (One of the two spaces preceding the second item is the result of conversion from an arithmetic data type to a character string.)

17.1.2.5 PUT PAGE

The PUT PAGE statement positions the output file at the start of a new page. This statement is valid only for print files, that is, files that have been opened with the PRINT attribute.

The form of the PUT PAGE statement is

PUT [FILE(file-reference)] PAGE;

file-reference

A reference to a print file that is to be advanced to the next output page. The defaults for the file are specified in Section 17.1.2.1.

The following example demonstrates the PUT PAGE statement:

```
PUT FILE(REPORT) PAGE SKIP LINE(2);
```

The PUT statement advances the file REPORT to the beginning of the next page, advances to line 2, and skips to the beginning of the next line (3).

17.1.2.6 PUT SKIP

The PUT SKIP statement positions the output file at the start of a new line.

The form of the PUT SKIP statement is

```
PUT [ FILE(file-reference) ] SKIP [(expression)];
```

file-reference

A reference to the file to which the SKIP option applies. The defaults for this file are specified in Section 17.1.2.1.

expression

An expression giving the number of lines to be advanced. It must yield an integer i, which must not be negative and must be greater than 0 (except for print files). If the expression is omitted, i equals 1. The processing performed by PUT SKIP is described in Section 17.1.2.1.

17.1.3 FORMAT Statement

The FORMAT statement describes a remote format-specification list to be used by GET EDIT or PUT EDIT statements. The FORMAT statement and remote (R) format item are convenient when the same format specification is used by a large number of GET EDIT and/or PUT EDIT statements. In such a case, any change in the format specification can be made in the single FORMAT statement, rather than in each GET or PUT statement.

The form of the FORMAT statement is

```
label: FORMAT (format-specification,...);
```

label

A valid PL/I label, which is required. It is specified in the GET EDIT or PUT EDIT statement that contains a remote format item, R, in its format-specification list.

format-specification

A list of one or more format items that match corresponding input targets in a GET EDIT statement, or output sources in a PUT EDIT statement. For more information, see Section 17.3.

17.2 Stream I/O Processing and Positioning

The following sections describe how PL/I positions a stream when the source or target of the stream is a stream file (Section 17.2.1) or a string

(Section 17.2.3). Section 17.2.2 describes special features of print files, which are stream files having the OUTPUT and PRINT attributes.

17.2.1 Processing and Positioning of Stream Files

A stream file is a file of ASCII text, divided into lines. For every stream file used in a program, PL/I maintains the following information:

- The locations of the beginning and end of the file. On input operations, the ENDFILE condition is signaled on the first attempt to read past the end of the file.
- For output files, the maximum number of ASCII characters in a line, or the line size. The line size is either a default value or the specific value you have established for the file. The default line size is as follows:
  - If the output is to a physical record-oriented device, such as a line printer or terminal, the default line size is the width of the device.
  - If the output is to a nonrecord device (disk), the default line size is 510.

  The line size is used to determine when to skip to the next line. On input, a single data item cannot cross a line unless it is a character string enclosed in apostrophes. On output, data items may be split across lines.
- The current position in the file. Essentially, this is the point at which the last input or output operation stopped. It is the exact character position (sometimes in the middle of a line) at which the next output item is written or from which the next input item is read.

Input operations can begin anywhere from the current position onward. The default is the current position. To acquire data from a different position, you can

- Use the SKIP option of the GET statement to advance by a specified number of lines before reading data.
- Use control format items to move to a specified position before reading data. With the GET statement, control format items are restricted to SKIP (same operation as the SKIP option), COLUMN (advance to a specified character position), and X (advance by a specified number of character positions from the current position). Note that the control format items, unlike the SKIP option, are executed during, not before, the input of data. (Section 17.3 describes format items.) The control format items can signal the ENDFILE and ERROR conditions if the end-of-file is encountered.

- Close and then reopen the file, to set the current position at the first character in the file.
- Use the REWIND built-in subroutine (see Section 16.4.5).

Because stream files are sequential, output operations always place data at the end of the file. You can do the following additional formatting of output with any stream output file:

- Use the SKIP option of the PUT statement to skip lines. The SKIP option inserts null lines in the file between the current position and the position of the next output. The SKIP option can reposition the file even though no data is output.
- Use the control format items to advance to a specified line or character position, or to a new page. The control format items are COLUMN (move to a specified character position), SKIP (same effect as the SKIP option), and X (skip a specified number of characters following the current position). As is the case with input, control format items are executed only during the output of data; if only part of the format list is used, the excess items are ignored.

17.2.2 Processing and Positioning of Print Files

A print file is a stream output file that is intended for output on a terminal, line printer, or other output device. Any stream output file can be declared a print file by use of the PRINT attribute. The default stream output file, SYSPRINT, is also a print file. Terminals should always be declared as print files when used for output.

The following list describes the special features of print files as opposed to ordinary stream output files.

- Character strings are not enclosed in apostrophes on list-directed output.
- List-directed output data items are separated by tabs instead of spaces. Tab stops occur at 8-column increments beginning with column 1. With the PUT EDIT statement and the TAB format item, you can begin output at a specified tab stop.
- Print files are divided into both lines and pages. A record is kept internally of the number of lines per page. You can specify a page size when the file is created.
- During output of data to a print file, the ENDPAGE condition is signaled when the output exceeds the page size.
- New pages are started by the PUT PAGE statement, the PAGE format item, and certain other format items. Each of these operations increments the current page number by 1.

- If the print file is a terminal, the output is written to it at the conclusion of each PUT statement.
- A print file is created with PRN-format carriage control. PRN format is efficient for both terminals and line printers because blank lines do not require individual records. (It is discussed in the *VAX-11 Record Management Services Reference Manual.*)
- Print files usually cannot be read properly with GET LIST or GET EDIT.

PL/I maintains several values for a print file. These values, and the ways you can use them, are

- The current page number. The first output to a print file is written to page 1. The current page number is incremented by the PAGE option, the PAGE format item, and, in some circumstances, by the LINE option and LINE format item. The PAGENO built-in function returns the current page number from a print file, thus allowing you to keep track of the number of pages being written to a file. You can set the current page number to a specific value by assigning the value to the PAGENO pseudovariable.
- The page size. This is an integer that specifies the number of lines on a page. The page size is either the default value or the specific number that you have established for the print file. The default page size is as follows:
  - If the logical name SYS$LP_LINES is defined, the default page size is the numeric value of SYS$LP_LINES - 6.
  - If SYS$LP_LINES is not defined, or if its value is less than 30 or greater than 99, or if its value is not numeric, the default page size is 60.

  When the last line on a page is filled, the first attempt to write (or position the file) beyond that position signals the ENDPAGE condition. It is signaled only on the first such attempt; PL/I executes a PUT PAGE unless an ON-unit is established for the condition. The ON-unit can, for example, write a trailer at the bottom of the current page, or a header at the top of the next one, before printing a new page of data.
- The current line number. This is an integer specifying the line currently being used for output, relative to the top of the page. The first line on the page is line 1. The LINENO built-in function can evaluate the current line of a specified print file. The LINE option of the PUT statement, and the LINE format item, can reposition the file to a specified line.

- Position of tab stops. The TAB format item can reposition a print file to a specified tab stop relative to the current position.

17.2.3 Processing and Positioning of Character Strings

If the input or output stream is a character string, the processing is similar to that of files, but the positioning options are more limited:

- Input can begin either at the beginning of the string or at a specified character position. The ERROR condition is signaled if the end of the string is encountered inside a quoted string, or if an attempt is made to read past the last input field or to read a null string. Only the X format item can be used for positioning.
- The first output by a PUT statement always occurs at the beginning of the string, and subsequent output by the same statement follows the previous output. The ERROR condition is signaled if the maximum length of the string is exceeded. Only the X format item can be used for positioning.

On input, the value of the character-string expression specified in the STRING option must include commas or spaces to separate input fields, as with any stream input. For an example, see Section 17.1.1.3.

17.3 Format Items and Specifications

This section describes the formatting of input and output data in GET EDIT and PUT EDIT statements. Section 17.3.1 describes the individual format items; Section 17.3.2 shows how to combine them into format specifications.

17.3.1 Format Items

There are three categories of PL/I format items:

- The data format items, A, B, E, F, and P, are used for input or output of data. A and B are used for character- and bit-string formats, respectively. E and F are used for floating- and fixed-point formats, respectively. P is used for input or output of data in a specified picture format. All data format items can be used with either the FILE or STRING option in edit-directed statements.
- The remote format item, R, is used to specify the label of a FORMAT statement, which contains a remote list of format items.
- The control format items, SKIP, LINE, PAGE, TAB, COLUMN, and X, are used to control the position in the input or output stream at

which data is placed or from which it is acquired. Only X can be used with the STRING option in edit-directed statements.

NOTE

Arguments for all format items, except picture (P) and remote (R), may be integer expressions.

Each data format item refers to a field of characters in the stream. It specifies the width of the field in characters and either the manner in which the field is used to represent a value (output) or the manner in which the characters in the field are to be interpreted (input). Because the representation or interpretation is under control of the format items, certain symbols used in the stream with GET LIST and PUT LIST are not used with GET EDIT or PUT EDIT:

- Strings input by the GET EDIT statement should not be enclosed in apostrophes unless they are intended to be part of the string. Strings output by PUT EDIT are not enclosed in apostrophes.
- Bit strings input by the GET EDIT statement should not be enclosed in apostrophes nor be followed by the radix factors B, B1, B2, B3, or B4. These factors are not added by the PUT EDIT statement on output.
- The comma and space characters are not interpreted as data separators on input. On output, values are not automatically separated by spaces.

The following guidelines apply to errors and mismatches that occur between the actual data values and the fields specified by data format items:

- On input, the ERROR condition is signaled if the field of characters cannot be interpreted as required by the format item.
- On output, strings are left justified in the specified field, and numeric data is right justified. Truncation occurs if the field is too narrow to contain the necessary characters. Strings are truncated on the right and numeric data on the left.

17.3.2 Format Specifications

In the GET EDIT, PUT EDIT, and FORMAT statements, format items are used singly or in combination to make up format specifications. A format specification can be written four ways:

```
format-item
(replication-factor)'format-item'
iteration-factor format-item
iteration-factor(format-specification,...)
```

The entire format specification must be enclosed in parentheses.

The iteration factor is an integer that repeats its subsequent format item or list of format specifications. If the iteration factor precedes a single format item not in parentheses, the iteration factor and format item must be separated by a space. For example, the statement

```
PUT EDIT (A) (F(5,2));
```

specifies a 5-character field containing decimal digits, two of which are fractional. Used by itself as a format specification, this item specifies one such field. To specify two such fields, you must precede the item with the iteration factor 2:

```
PUT EDIT (A,B) (2 F(5,2));
```

As shown by the third form above, an iteration factor can also repeat an entire list of format specifications, as in

```
PUT EDIT ( (A(I) DO I = 1 TO 10) ) /* 10 array elements */
( 2( F(5,2),2(F(7,2),E(8)) ) );     /* 10 format items */
```

Expanded into individual format items, this specification is

```
F(5,2),F(7,2),E(8),F(7,2),E(8),
F(5,2),F(7,2),E(8),F(7,2),E(8)
```

When PL/I performs an edit-directed operation, it examines the list of input targets or output sources, beginning with the first. If the target or source is an array, the array is expanded in row-major order to form an ordered list of individual data items. (For example, in the PUT EDIT statement above, you could simply write PUT EDIT (A) followed by the format specification; the DO clause is not necessary.) If the target or source is a structure, the structure is expanded in the order of its declaration to form a list of individual items. If the target or source contains a DO specification, the item or items that precede the DO keyword are expanded in the preceding manner, and an ordered list of individual items is then created as per the DO specification.

Within a single target or source, items at the deepest level of parentheses are processed first.

Table 17-1: Summary of Format Items

Format Item	Use
A[(w)]	With GET EDIT, reads w characters from the input stream; with PUT EDIT, converts the value to be output to a w-character string and outputs the resulting string. If w is omitted with PUT EDIT, the field width equals the length of the converted output source. You cannot omit w with GET EDIT.

Table 17-1 (Cont.): Summary of Format Items

Format Item	Use
B[(w)]	With GET EDIT, reads w binary digits (0s and 1s) from the input stream; with PUT EDIT, the corresponding value is first converted to a BIT value (if necessary), then converted to a character string of length w, containing 0s and 1s, and written to the output stream. The B format item is equivalent to B1. If w is omitted with PUT EDIT, the field width equals the length of the converted output source. You cannot omit w with GET EDIT.
Bn[(w)]	With GET EDIT, reads a character string of length w from the input stream; with PUT EDIT, the corresponding value is first converted to a BIT value (if necessary), then converted to a character string of length w and written to the output stream. The base and allowable characters are controlled by n, which is the power of 2 in which the value is represented. B1 is equivalent to B and results in binary values. B2 results in base-4 values; B3 in octal values; B4 in hexadecimal values. The characters allowed for each value of n are: *n Characters* 1 0,1 2 0–4 3 0–7 4 0–9,A–F If w is omitted with PUT EDIT, the field width equals the length of the converted output source. You cannot omit w with GET EDIT.
COLUMN(position)	With GET EDIT, specifies the position at which reading of data is to proceed; with PUT EDIT, outputs spaces until the specified column position. May be used with files only.
E(w[,d])	With GET EDIT, converts a field of w characters from the input stream to a floating-point number; with PUT EDIT, converts a value to a w-character floating-point representation with d fractional digits in the mantissa and writes the w-character string to the output stream. If d is omitted on output, all fractional digits are written out. If d is omitted on input, it is assumed to be zero (no fractional digits). If the input value contains a decimal point, the value of d is ignored.
F(w[,d])	With GET EDIT, converts a field of w characters from the input stream to a fixed-point value; with PUT EDIT, converts a value to a w-character fixed-point representation with d fractional digits, and writes the w-character string to the output stream. If d is omitted with GET EDIT, it is assumed to be zero, although a decimal point in the source overrides a specification of d. If d is omitted with PUT EDIT, it is assumed to be zero; only the integral part of the number (without a decimal point) appears in the output stream. (If d is specified with PUT EDIT, d plus the number of integral digits must not exceed 31.)
LINE(number)	Valid for print files only. Specifies a line number, relative to the top of the page, at which output is to continue.

Table 17-1 (Cont.): **Summary of Format Items**

Format Item	Use
P´picture	With GET EDIT, acquires a character string from the input stream whose length is specified by the picture specification, and signals ERROR if the string is not a valid pictured value; with PUT EDIT, converts an expression to a pictured value as specified by the picture, and writes the pictured value to the output stream.
PAGE	Valid for print files only. Specifies that output is to be continued at the top of the next page.
R(label)	Indicates that format items are to be acquired from the FORMAT statement at the specified label.
SKIP[(linecount)]	With GET EDIT, continues reading after ´linecount´ lines; with PUT EDIT, outputs ´linecount´ blank lines and continues output. May be used with files only. If omitted, linecount defaults to 1.
TAB[(n)]	Valid for print files only. Continues output at the nth tab stop relative to the current position.
X[(n)]	With GET EDIT, ignores n characters in the input stream; with PUT EDIT, places n spaces in the output stream. May be used with either files or character strings. If n is omitted (permissible only with PUT EDIT), it defaults to 1.

Given a list of one or more data items contained in the first target or source, PL/I processes the data items from left to right. Beginning with the leftmost data item, and for each subsequent item, PL/I executes format items until the data item has been either assigned a value from the input stream or converted to a character representation and placed in the output stream. Control format items are therefore executed in the order in which they occur in the format-specification list. With the first target or source, the execution of format items begins with the leftmost format item in the list. If the end of the list is reached, PL/I returns to the leftmost format item and continues.

When all items contained in the first target or source have been processed, PL/I evaluates the next target or source, then examines the format-specification list, beginning where the previous operation stopped. This processing continues until all data items in the input-target or output-source list have been processed, at which point the edit-directed statement terminates. If termination occurs while PL/I is in the middle of the list of format items, those items to the right of the termination point are not executed.

Chapter 18

Record Input/Output

Record I/O is performed by the READ, WRITE, DELETE, and REWRITE statements; each statement processes an entire record. (In stream I/O, more than one line or record can be processed by a single statement.) In addition, some forms of record I/O allow you to access records in the file by record number or by a key field contained in the record.

This chapter describes the following topics:

- Statements used for record I/O
- The organization and use of sequential files, including files on magnetic tape
- The organization and use of relative files
- The organization and use of indexed sequential files

Table 18–1 contains a summary of attributes for record files.

18.1 Statements for Record I/O

Sections 18.1.1 through 18.1.4 describe the READ, WRITE, REWRITE, and DELETE statements, and provide some examples of their use. Section 18.1.5 describes their options. Sections 18.2, 18.3, and 18.4 contain examples of the statements in use with sequential, relative, and indexed sequential files, respectively.

For an open record file, PL/I maintains the following position information:

- The next record, for files with the SEQUENTIAL INPUT or SEQUENTIAL UPDATE attributes. It designates the record to be accessed by a READ statement that does not specify the KEY option. The next record may contain end-of-file.
- The current record, for a file with the UPDATE attribute. It designates either of the following:
  - The record to be modified by a REWRITE statement that does not specify the KEY option
  - The record to be deleted by a DELETE statement that does not specify the KEY option

  The value of the current record may be undefined.

Table 18-1: Attributes and Access Modes for Record Files

Attributes Specified	Attributes Implied	Valid Devices and File Organizations	Usage
SEQUENTIAL OUTPUT	RECORD	Any output device or file except indexed	Records may be added to the end of the file using WRITE statements. Each WRITE statement adds a single record to the file.
SEQUENTIAL INPUT	RECORD	Any input device or file	Records in the file are read using READ statements. Each statement reads a single record.
SEQUENTIAL UPDATE	RECORD	Relative, indexed, sequential disk	READ statements read a file's records in order. PL/I maintains the current record, which is the record just read. This record may be replaced in a REWRITE statement.[1] In a relative or indexed sequential file, the current record may also be deleted with a DELETE statement. Each statement processes a single record.
DIRECT OUTPUT	KEYED RECORD	Relative, indexed, sequential disk[2]	WRITE statements insert records into the file at positions specified by keys. Each statement inserts a single record.
DIRECT INPUT	KEYED RECORD	Relative, indexed, sequential disk[1]	READ statements specify records to be read randomly by key. Each statement reads a single record.
DIRECT UPDATE	KEYED RECORD	Relative, indexed, sequential disk[1]	READ, WRITE, and REWRITE statements specify records randomly by key. In a relative or indexed file, records may also be deleted by key.

1. For a file with sequential organization, the record being written must have the same length as the one that was read.
2. The file must have fixed-length records.

Table 18-1 (Cont.): Attributes and Access Modes for Record Files

Attributes Specified	Attributes Implied	Valid Devices and File Organizations	Usage
KEYED SEQUENTIAL OUTPUT	RECORD	Relative, indexed, sequential disk[2]	WRITE statements insert records into the file at positions specified by keys. Each statement inserts a single record. This mode is identical to DIRECT OUTPUT.
KEYED SEQUENTIAL INPUT	RECORD	Relative, indexed, sequential disk[1]	READ statements access records in the file randomly by key or sequentially.
KEYED SEQUENTIAL UPDATE	RECORD	Relative, indexed, sequential disk[1]	Any record I/O operation is allowed except a WRITE statement that does not specify a key or a DELETE statement for a sequential disk file with fixed-length records.

1. For a file with sequential organization, the record being written must have the same length as the one that was read.
2. The file must have fixed-length records.

When a file is opened, the current record is undefined and the next record designates the first record in the file or, if the file is empty, end-of-file. After a sequential read, the current record designates the record just read; the next record indicates the following record or, if there are no more records, end-of-file.

After a keyed I/O statement, that is, one that specifies the KEY or KEYFROM option, the current record and next record are set as follows:

Statement	Current Record	Next Record
READ	X	X+1
WRITE	X	X+1
REWRITE	X	X+1
DELETE	undefined	X+1

where X is the record specified by key, and X+1 is the next record or, if there are no more records, the end-of-file.

18.1.1 READ Statement

The READ statement reads a record from a file, either the next record or a record specified by the KEY option. The file must have either the INPUT or the UPDATE attribute. The format of the READ statement is

```
READ FILE (file-reference)

          { INTO (variable-reference) }
          { SET (pointer-variable)    }

          [ KEY (expression)           ]
          [ KEYTO (variable-reference) ]

          [ OPTIONS (option,...) ];
```

file-reference

Specifies the file from which the record is to be read. If the file is not currently open, PL/I opens it with the implied attributes RECORD and, if it lacks the UPDATE attribute, INPUT. The implied attributes are merged with those specified in the file's declaration (see Section 16.1.2).

INTO (variable-reference)

Specifies that the contents of the record are to be assigned to the specified variable. The variable must be byte addressable.

If the variable has the VARYING attribute and the file does not have the attribute ENVIRONMENT(SCALARVARYING), the entire rec-

ord is treated as a string value and assigned to the variable; if the record is longer than the variable, it is truncated and the ERROR condition is signaled. For any other type of variable, the record is simply copied into the variable's storage. If the record is not exactly the same size as the target variable, as much of the record as will fit is copied into the variable and the ERROR condition is signaled.

SET (pointer-variable)

Specifies that the record should be read into a buffer allocated by PL/I and that the specified pointer variable be assigned the value of the location of the buffer in storage.

This buffer remains allocated until the next operation on the file, but no longer. Therefore, neither the pointer value nor the buffer should be used after that operation. The only valid use of the buffer during a subsequent I/O operation is in a REWRITE statement, in which case, the record is rewritten from the buffer before it is deallocated.

KEY (expression)

Specifies that the record to be read is to be located using the key specified by the expression. The file must have the KEYED attribute. The key value must have a computational data type.

The nature of the key depends on the file's organization, as follows:

- For a relative file or a sequential disk file with fixed-length records, the key is a fixed binary value indicating the relative record number of the record to be read.
- For an indexed sequential file, the key specifies a key contained within a record. The data type of the key and its location within the record are as specified when the file was created.

The value of the specified expression is converted to the data type of the key. If no record with the specified key exists in the file, or if the value specified is not valid for conversion to the data type of the key, the KEY condition is signaled.

KEYTO (variable-reference)

Specifies that the key of the record being read is to be assigned to the designated variable. The value of the key is converted from the data type implied by the file's organization to that of the variable. The variable must have a computational data type, but cannot be an unaligned bit string or an aggregate consisting entirely of unaligned bit strings.

KEYTO can be specified only for a file that has both the KEYED and SEQUENTIAL attributes. It conflicts with the KEY option.

OPTIONS (option,...)

Specifies one or more of the following READ statement options, separated by commas:

FIXED_CONTROL_TO (variable-reference)
INDEX_NUMBER (expression)
MATCH_GREATER
MATCH_GREATER_EQUAL
RECORD_ID (variable-reference)
RECORD_ID_TO (variable-reference)

These options are summarized in Section 18.1.5.

If the file is accessed sequentially, the READ statement reads the next record. If the next record position is at the end-of-file, the ENDFILE condition is signaled.

After a successful read, the current record position denotes the record that was just read. The next record position denotes the following record or, if there is none, end-of-file.

If any error occurs other than an incorrect record size, the current record becomes undefined and the next record remains as it was before the read was attempted.

The examples below demonstrate the READ statement.

The following program illustrates reading a file with variable-length records into a character string with the VARYING attribute, and writing the records to a new output file:

```
COPY: PROCEDURE;
DECLARE INREC CHARACTER(80) VARYING,
        ENDED BIT(1) STATIC INIT('0'B),
        (INFILE,OUTFILE) FILE;

        OPEN FILE (INFILE) RECORD INPUT
                TITLE('RECFILE.DAT');

        OPEN FILE (OUTFILE) RECORD OUTPUT
                TITLE('COPYFILE.DAT');

        ON ENDFILE(INFILE) ENDED = '1'B;

        READ FILE(INFILE) INTO (INREC);
        DO WHILE (^ENDED);
                WRITE FILE (OUTFILE) FROM (INREC);
                READ FILE (INFILE) INTO (INREC);
                END;
        CLOSE FILE(INFILE);
        CLOSE FILE(OUTFILE);
        RETURN;
        END;
```

The procedure COPY uses a DO-group to read the records in the file sequentially until the end-of-file is reached. It uses the ON statement to establish the action to be taken when the end-of-file occurs: it sets the bit ENDED to '1'B so that the DO-group will not execute again.

The VARYING character-string variable INREC has a maximum length of 80 characters. If any record in the file is longer than that, the ERROR condition is signaled. If no ERROR ON-unit exists, the program exits.

The next example shows a keyed READ statement accessing a record in an indexed sequential file:

```
DECLARE 1 STATE,
          2 NAME CHARACTER(30),
          2 CAPITAL,
            3 NAME CHARACTER(20),
               .
               .
               .
          2 SYMBOLS,
            3 FLOWER CHARACTER(30),
            3 BIRD CHARACTER(30),
        STATE_FILE FILE,
        INPUT_NAME CHARACTER(30) VARYING;
            .
            .
            .
OPEN FILE(STATE_FILE) KEYED;
    PUT SKIP LIST('State?');
    GET LIST(INPUT_NAME);
    READ FILE(STATE_FILE) INTO(STATE)
            KEY(INPUT_NAME);
    PUT SKIP LIST('The flower of',STATE.NAME,
                 'is the',FLOWER);
```

The file STATE_FILE is opened for keyed access, and the READ statement specifies the key of interest in the KEY option. The value for this option is determined at run time by a GET statement. In the READ statement, the contents of a record from the file STATE_FILE are read into the structure STATE.

The next example illustrates accessing a relative file sequentially with READ statements and obtaining the key value of each record, that is, the relative record number:

```
PRINT_DATA: PROCEDURE OPTIONS(MAIN);

DECLARE 1 EMPLOYEE BASED (EP),
          2 NAME,
            3 LAST CHAR(30),
            3 FIRST CHAR(20),
            3 MIDDLE_INIT CHAR(1),
          2 DEPARTMENT CHAR(4),
          2 SALARY FIXED DECIMAL (6,2),
        EP POINTER,
        EMP_FILE FILE;
```

```
DECLARE EOF BIT(1) STATIC INIT('0'B),
        NUMBER FIXED BIN(31);

        ON ENDFILE(EMP_FILE) EOF = '1'B;
        OPEN FILE(EMP_FILE) INPUT SEQUENTIAL KEYED;

        READ FILE(EMP_FILE) SET(EP) KEYTO(NUMBER);
        DO WHILE (^EOF);
           PUT SKIP LIST('EMPLOYEE',NUMBER,
                NAME.FIRST,NAME.LAST,MIDDLE_INIT);
           READ FILE(EMP_FILE) SET(EP) KEYTO(NUMBER);
           END;
        CLOSE FILE(EMP_FILE);

END;
```

The records in the file EMP_FILE are arranged according to employee numbers, each of which corresponds to a relative record number in the file. READ statements read records into the based structure EMPLOYEE and set the pointer EP to the location of the allocated buffer. The READ statements specify the KEYTO option to obtain the record number of each record. The procedure prints the employee numbers and names. When the last record has been read, the program closes the input file and exits.

18.1.2 WRITE Statement

The WRITE statement adds a record to a file, either at the end of a file that has the SEQUENTIAL and OUTPUT attributes, or in a specified key position in a file that has the KEYED and OUTPUT attributes or the KEYED and UPDATE attributes. The format of the WRITE statement is

```
WRITE FILE(file-reference) FROM (variable-reference)

            [ KEYFROM (expression) ]
            [ OPTIONS (option,...) ];
```

file-reference

A reference to the file to which the record is to be written. If the file is not currently open, the WRITE statement opens it with the implied attributes RECORD, OUTPUT, and SEQUENTIAL; they are merged with the attributes specified in the file's declaration (see Section 16.1.2).

variable-reference

A reference to the variable containing data for the output record. The variable must be byte addressable.

If the variable has the VARYING attribute, and the file does not have the attribute ENVIRONMENT(SCALARVARYING), the WRITE statement writes only the current value of the varying string into the specified record. In all other cases, the WRITE statement writes the entire storage of the variable. If the contents of the variable do not fit the specified record size, the WRITE statement outputs as much of the variable as will fit, and the ERROR condition is signaled.

KEYFROM (expression)

An option specifying that the record to be written is to be positioned in the file according to the key specified by expression. KEYFROM is required if the file has the KEYED attribute and invalid if it does not.

The nature of the key depends on the file's organization, as follows:

- For a relative file or a sequential disk file with fixed-length records, the key value is fixed binary and indicates the relative record number of the record to be written.
- For an indexed sequential file, the key specifies the record's primary key. PL/I inserts the key value specified into the correct key field in the record and sets the key number to the primary index.

The value of the specified expression is converted to the data type of the key. If a record with the specified key already exists, or if the value specified cannot be converted to the data type of the key, the KEY condition is signaled.

OPTIONS (option,...)

An option specifying one or both of the following WRITE statement options, separated by commas:

FIXED_CONTROL_FROM (expression)
RECORD_ID_TO (variable-reference)

These options are summarized in Section 18.1.5.

If the file has the UPDATE attribute, the current record is set to designate the record just written. The next record is set to designate the following one; if there is none, the next record is set to end-of-file.

The examples below demonstrate the WRITE statement.

The program TRUNC reads a file with variable-length records into a character string with the VARYING attribute, and creates a sequential output file in which each record has a fixed length of 80 characters.

```
TRUNC: PROCEDURE;
DECLARE INREC CHARACTER(80) VARYING,
        OUTREC CHARACTER(80),
        ENDED BIT(1) STATIC INIT('0'B),
        (INFILE,OUTFILE) FILE;

        OPEN FILE (INFILE) RECORD INPUT
                TITLE('RECFILE.DAT');
        OPEN FILE (OUTFILE) RECORD OUTPUT
                TITLE('TRUNCFILE.DAT')
                ENVIRONMENT(FIXED_LENGTH_RECORDS,
                        MAXIMUM_RECORD_SIZE(80));

        ON ENDFILE(INFILE) ENDED = '1'B;

        READ FILE(INFILE) INTO (INREC);
        DO WHILE (^ENDED);
                OUTREC = INREC;
                WRITE FILE (OUTFILE) FROM (OUTREC);
                READ FILE (INFILE) INTO (INREC);
                END;
        CLOSE FILE(INFILE);
        CLOSE FILE(OUTFILE);
        RETURN;
        END;
```

The ENVIRONMENT attribute for the file OUTFILE specifies the record format and length of each fixed-length record. When records are written to a file with fixed-length records, the variable specified in the FROM option must have the same length as the records in the target output file. Otherwise, the ERROR condition is signaled. Thus, in the above example, each record read from the input file is copied into a fixed-length character-string variable for output.

Each time this program is executed, it creates a new version of the file TRUNCFILE.DAT.

The next example adds records to the existing relative file EMP_FILE. The file is organized by employee numbers, with each record occupying the relative record number in the file that corresponds to the employee number.

```
ADD_EMPLOYEE: PROCEDURE;

DECLARE 1 EMPLOYEE,
          2 NAME,
            3 LAST CHAR(30),
            3 FIRST CHAR(20),
            3 MIDDLE_INIT CHAR(1),
          2 DEPARTMENT CHAR(4),
          2 SALARY FIXED DECIMAL (6,2),
        EMP_FILE FILE;
```

```
DECLARE MORE_INPUT BIT(1) STATIC INIT('1'B),
        NUMBER FIXED DECIMAL (5,0);

        OPEN FILE(EMP_FILE) DIRECT UPDATE;

        DO WHILE (MORE_INPUT);
          PUT SKIP LIST('Employee Number');
          GET LIST (NUMBER);
          PUT SKIP LIST
           ('Name (Last, First, Middle Initial)');
          GET LIST
           (EMPLOYEE.NAME.LAST,EMPLOYEE.NAME.FIRST,
               EMPLOYEE.NAME.MIDDLE_INIT);

          PUT SKIP LIST('Department');
          GET LIST (EMPLOYEE.DEPARTMENT);

          PUT SKIP LIST('Starting salary');
          GET LIST(EMPLOYEE.SALARY);

          WRITE FILE (EMP_FILE)
               FROM (EMPLOYEE) KEYFROM(NUMBER);

          PUT SKIP LIST('More (0 or 1)?');
          GET LIST(MORE_INPUT);
          END;
        CLOSE FILE(EMP_FILE);
        RETURN;
        END;
```

The file is opened with the DIRECT and UPDATE attributes, since records will be written only by referring to a key number. Within the DO-group, the program prompts for data for each new record that will be written to the file. After the data is input, the WRITE statement specifies the KEYFROM option to provide the relative record number. The number itself is not a part of the record, but will be used to retrieve the record when the file is accessed for keyed input.

18.1.3 REWRITE Statement

The REWRITE statement replaces a record in a file, either the current record or one specified by the KEY option. The file must have the UPDATE attribute. The format of the REWRITE statement is

```
REWRITE FILE (file-reference)

   [ FROM (variable-reference) [ KEY (expression) ]]
   [ OPTIONS (option,...) ];
```

file-reference

A reference to the file that contains the record to be replaced. If the file is not open, it is opened with the implied attributes RECORD and

UPDATE; they are merged with the attributes specified in the file's declaration (see Section 16.1.2).

FROM (variable-reference)

An option giving the variable whose value is to be used to rewrite the specified record. The variable must be byte addressable.

If the FROM option is not specified, there must be a currently allocated buffer from an immediately preceding READ statement that specified the SET option, and the file must have the SEQUENTIAL attribute. In this case, the record is rewritten from the buffer containing the record that was read.

If the variable has the VARYING attribute, and the file does not have the attribute ENVIRONMENT(SCALARVARYING), the REWRITE statement writes only the current value of the varying string into the specified record. The size of the new record must be the same as the size of the old record, or an RMS error will occur. In all other cases, the REWRITE statement writes the variable's entire storage.

KEY (expression)

An option specifying that the key specified by expression will locate the record to be rewritten. This option is required if the file has the KEYED or DIRECT attribute. The expression must have a computational data type. The FROM option must be specified.

The nature of the key depends on the file's organization, as follows:

- For a relative file or a sequential disk file with fixed-length records, the key is a fixed binary value indicating the relative record number of the record to be rewritten.
- For an indexed sequential file, the key specified is contained within a record. The data type of the key and its location within the record are as specified when the file was created. The primary key field in the record cannot be modified.

The value of the specified expression is converted to the data type of the key. If no record with the specified key exists, if the value specified is not valid for conversion to the data type of the key, or if the primary key in a record in an indexed sequential file has been modified, the KEY condition is signaled.

OPTIONS (option,...)

An option giving one or more of the REWRITE statement options listed below, separated by commas:

FIXED_CONTROL_FROM(expression)
INDEX_NUMBER (expression)

MATCH_GREATER
MATCH_GREATER_EQUAL
RECORD_ID (variable-reference)
RECORD_ID_TO (variable-reference)

These options are summarized in Section 18.1.5.

After execution of the REWRITE statement, the next record position is set to denote the record immediately following the one rewritten or, if there is none, end-of-file. The current record is set to designate the record just rewritten. For disk files with sequential organization, the record rewritten must have the same length as the one that was read.

The examples below demonstrate the REWRITE statement.

The procedure NEW_SALARY updates the salary field in a relative file containing employee records. The procedure receives two input parameters: the employee number and the new salary. The employee number is the key value for the records in the relative file.

```
NEW_SALARY: PROCEDURE (EMPLOYEE_NUMBER,PAY);

DECLARE EMPLOYEE_NUMBER FIXED DECIMAL(5,0),
        PAY FIXED DECIMAL (8,2);

DECLARE 1 EMPLOYEE,
          2 NAME,
            3 LAST CHAR(30),
            3 FIRST CHAR(20),
            3 MIDDLE_INIT CHAR(1),
          2 DEPARTMENT CHAR(4),
          2 SALARY FIXED DECIMAL (8,2),
        EMP_FILE FILE;

        OPEN FILE(EMP_FILE) DIRECT UPDATE;
        READ FILE(EMP_FILE) INTO(EMPLOYEE)
                KEY (EMPLOYEE_NUMBER);
        EMPLOYEE.SALARY = PAY;
        REWRITE FILE(EMP_FILE) FROM(EMPLOYEE)
                KEY(EMPLOYEE_NUMBER);
        CLOSE FILE(EMP_FILE);
        RETURN;
        END;
```

The KEY option is specified in the READ statement, which obtains the record of interest, and in the REWRITE statement, which replaces the record with its new information in the file. Note that the FROM and KEY options must both be specified on the REWRITE statement.

The program CHANGE_HEADER changes the contents of the first record in the sequentially organized file TITLE_PAGE. The file consists of 80-byte, fixed-length records.

```
CHANGE_HEADER: PROCEDURE OPTIONS(MAIN);

DECLARE TITLE_PAGE FILE SEQUENTIAL UPDATE,
        INREC CHARACTER(80) BASED(P),
        P POINTER;

        OPEN FILE(TITLE_PAGE);
        READ FILE(TITLE_PAGE) SET(P);

        INREC = 'Summary of Courses for Fall 1983';
        REWRITE FILE(TITLE_PAGE);
        CLOSE FILE(TITLE_PAGE);
        RETURN;
        END;
```

The READ statement specifies the SET option. The input record is read into a buffer, INREC, which is a based character-string variable. The assignment statement modifies the buffer, and the REWRITE statement rewrites the record. Because the REWRITE statement does not specify a FROM option, PL/I uses the contents of the buffer to rewrite the current record in the file (that is, the one just read).

18.1.4 DELETE Statement

The DELETE statement deletes a record from a file: the current record, the one specified by the KEY option, or the record specified by the RECORD_ID option. The file must have the UPDATE attribute. The format of the DELETE statement is

```
DELETE FILE(file-reference) [ KEY (expression) ]
            [ OPTIONS(option,...) ];
```

file-reference

A reference to the file from which the specified record is to be deleted. If the file is not currently opened, PL/I opens it with the implied attributes RECORD and UPDATE; they are merged with the attributes specified in the file's declaration (see Section 16.1.2).

KEY (expression)

An option specifying that the key specified by expression will be used to locate the record to be deleted. The file must have the KEYED attribute.

The nature of the key depends on the file's organization, as follows:

- For a relative file, the key is a fixed binary value indicating the relative record number of the record to be deleted.
- For an indexed sequential file, the key is contained in the record; its position in the record and its data type are as determined when the file was created.

The value of the specified expression is converted to the data type of the key. If no record with the specified key exists in the file, or if the value specified is not valid for conversion to the data type of the key, the KEY condition is signaled.

OPTIONS(option,...)

An option giving one or more of the DELETE statement options listed below, separated by commas:

FAST_DELETE
INDEX_NUMBER (expression)
MATCH_GREATER
MATCH_GREATER_EQUAL
RECORD_ID (variable-reference)

These options are summarized in Section 18.1.5.

After execution of the DELETE statement, the next record is set to denote the record following the deleted one. The current record is undefined.

The program BAD_RECORD, below, deletes an erroneous record in an indexed sequential file containing data about states. The primary key in the file is the name of a state.

```
BAD_RECORD: PROCEDURE OPTIONS(MAIN);

DECLARE STATE_FILE FILE KEYED UPDATE;
        OPEN FILE(STATE_FILE) TITLE('STATEDATA.DAT');
        DELETE FILE(STATE_FILE) KEY('ArKlansas');
        CLOSE FILE(STATE_FILE);
RETURN;
END;
```

The file is opened with the UPDATE attribute, and the OPEN statement specifies the file from which the record is to be deleted.

18.1.5 Options for Record I/O Statements

The options described in this section can be used with one or more of the record I/O statements. The description of each statement lists the options that are applicable to it.

FAST_DELETE

The FAST_DELETE option specifies, for a record in an indexed sequential file with alternate indexes, that only the primary index be updated. The alternate indexes for the deleted record are not updated until access to the record is attempted through the alternate index. This option applies only to indexed sequential files; it can improve the speed of deletions when such files are updated.

FIXED_CONTROL_FROM (expression)

The FIXED_CONTROL_FROM option specifies a value to be written in the fixed control portion of a record in a file with variable-length records and a fixed control area. The expression can be a scalar or a connected aggregate variable. It must not be an unaligned bit string or an aggregate consisting entirely of unaligned bit-string variables.

Observe the following rules when using this option:

- The file must have variable-length records with a fixed-length control area and must be opened with the OUTPUT or UPDATE attribute. With OUTPUT, the ENVIRONMENT option FIXED_CONTROL_SIZE must also be specified.
- The length of the expression must match the length of the fixed control area, as specified in the FIXED_CONTROL_SIZE option of ENVIRONMENT. If the variable is not of the correct length, the ERROR condition is signaled.

FIXED_CONTROL_TO (variable-reference)

The FIXED_CONTROL_TO option specifies that the contents of the fixed control area of a record in a file with a fixed control area are to be assigned to a variable specified by the variable-reference. The variable can be scalar or a connected aggregate. It must not be an unaligned bit string or an aggregate consisting entirely of unaligned bit-string variables.

Observe the following rules when using this option:

- The file must have variable-length records with a fixed-length control area, must be opened with the INPUT or UPDATE attributes, and should have the ENVIRONMENT option FIXED_CONTROL_SIZE_TO specified.
- For an existing file, the length of the variable must match the length of the fixed control area. If the length is not correct, the ERROR condition is signaled.

INDEX_NUMBER (integer-expression)

The INDEX_NUMBER option specifies the particular index in an indexed sequential file to which a KEY option applies (primary index, secondary index, and so on). The integer-expression specifies the index to use. The value of the expression must be the number of an index in the file. The primary index is 0, the secondary index is 1, and so on. The KEY option must also be specified on the statement.

The INDEX_NUMBER option on an I/O statement overrides the current index number, which may be set explicitly by the INDEX_NUMBER option of ENVIRONMENT or implicitly by a WRITE statement that specifies the KEY option or by a READ, REWRITE, or DELETE statement that specifies the RECORD_ID option.

When the INDEX_NUMBER option is used, the specified index becomes the current index for the file; it is used in this and in all subsequent I/O operations until the INDEX_NUMBER option is again changed. For example:

```
GET LIST(BIRD) OPTIONS (PROMPT('Enter bird:'));
READ FILE(STATEFILE) INTO(STATE) KEY(BIRD)
          OPTIONS (INDEX_NUMBER(2));
```

In this example, the READ statement accesses the record in the file STATEFILE using the second alternate index.

MATCH_GREATER

The MATCH_GREATER option indicates that the record of interest is the first record whose key is greater than that specified in the KEY option. MATCH_GREATER overrides the default rule for key matching, which is to look for an exact match.

Observe the following rules when using this option:

- The KEY option must also be specified.
- The file must be an indexed sequential or relative file.
- MATCH_GREATER conflicts with the MATCH_GREATER_EQUAL option.

MATCH_GREATER remains in effect only for the current statement. On subsequent accesses of the file, the default rule is in effect, unless you specify MATCH_GREATER again or MATCH_GREATER_EQUAL.

MATCH_GREATER_EQUAL

The MATCH_GREATER_EQUAL option indicates that the record of interest is the record whose key matches the one specified in the KEY option or, if no match is found, the first record whose key is greater than the one specified.

Observe the following rules when using this option:

- The KEY option must also be specified.
- The file must be an indexed sequential or relative file.
- MATCH_GREATER_EQUAL conflicts with the MATCH_GREATER option.

MATCH_GREATER_EQUAL remains in effect only for the current statement. On subsequent keyed accesses of the file, the default rule (an exact key match) is in effect, unless you specify MATCH_GREATER_EQUAL again or MATCH_GREATER.

RECORD_ID (variable-reference)

The RECORD_ID option indicates that the record of interest is specified by its record identification. The variable reference specifies the name of a 2-element array variable containing the record identification. The variable must be declared as (2) FIXED BINARY(31) and must be a connected array.

Observe the following rules when using this option:

- RECORD_ID conflicts with the KEY option on the READ, REWRITE, or DELETE statement.
- The file on which the operation is being performed must have been opened with the RECORD_ID_ACCESS option in the ENVIRONMENT attribute.
- If RECORD_ID is specified for an operation on an indexed sequential file, the RECORD_ID option resets the value of the current index number to 0 (the primary key).

RECORD_ID_TO (variable-reference)

The RECORD_ID_TO option specifies the name of a variable to be assigned the value of the record identification of the record on which the current operation is being performed. The variable reference denotes a 2-element array variable to receive the value of the record's identification. The variable must be declared as (2) FIXED BINARY(31), and it must be connected. The file on which the operation is being performed must have been opened with the RECORD_ID_ACCESS option of the ENVIRONMENT attribute.

18.2 Sequential Files

This section gives examples of some typical I/O operations on sequential disk files and on sequential devices, including magnetic tapes.

18.2.1 Creating a Sequential File

Whenever a PL/I program opens a file with the SEQUENTIAL OUTPUT attributes, VAX-11 PL/I normally creates a new sequential file. By default, records are variable length with a maximum length of 510 bytes. Each WRITE statement adds a new record to the file.

In VAX-11 PL/I, you can open a file with the APPEND option of ENVIRONMENT to add new records to the end of an existing sequential

file. This overrides the default action of PL/I, which is to create a new version when an existing file is opened for output. For example:

```
OPEN FILE(BIRD_FILE) OUTPUT SEQUENTIAL
               TITLE('BIRDS.DAT') ENV(APPEND);
WRITE FILE(BIRD_FILE) FROM (NEWDATA);
```

This OPEN statement opens the file BIRD_FILE and positions it at its current end-of-file. The WRITE statement then adds a new record at the end of the file.

18.2.2 Using Magnetic Tape Files

Before you execute a PL/I program that performs I/O to a file on a magnetic tape volume, you must use the following VAX/VMS operating system command language (DCL) commands:

1. Use the ALLOCATE command to allocate a device on which to mount a tape volume. For example:

   ```
   $ ALLOCATE MT:
     _MTA0: ALLOCATED
   ```

   The ALLOCATE command responds with the name of the physical device, on which you can now place the physical tape reel.

2. If the tape is new and you are going to write or overwrite, use the INITIALIZE command to format the tape and write a label on it. For example:

   ```
   $ INITIALIZE MTA0: MYTAPE
   ```

   This command writes the label MYTAPE on the tape volume mounted on MTA0:.

3. Use the MOUNT command to ready the volume for use and, optionally, to define a logical name for the device and file. For example:

   ```
   $ MOUNT MTA0: MYTAPE TAPEFILE
   ```

   After this sequence of commands, a PL/I program that writes records using the logical name TAPEFILE will be writing to the tape volume mounted on the device MTA0:. For example:

   ```
   DECLARE OUTFILE FILE RECORD OUTPUT;
   OPEN FILE(OUTFILE) TITLE('TAPEFILE:TAPE1.FIL');
   ```

   When this OPEN statement executes, the logical name TAPEFILE is translated to its equivalence MTA0:, and the open creates the file MTA0:TAPE1.FIL;0. Magnetic tape files always have a version number of 0.

18.2.2.1 Format of Magnetic Tapes

VAX-11 RMS (Record Management Services) supports the magnetic tape structure defined by ANSI X3.27-1977, the Magnetic Tape Labels and File Structure for Information Interchange. The tapes are encoded in ASCII format and can be processed on 9-track tape drives only. The INITIALIZE command writes a label on the tape in the required format. A tape created on a VAX/VMS system can be read on another system that supports the same tape label format.

18.2.2.2 Multivolume Tape Files

The ANSI standard X3.27-1977 for magnetic tapes allows any of the following combinations of tape files:

- A single file on a single volume (that is, a reel)
- A single file on more than one volume
- Multiple files on a single volume
- Multiple files on more than one volume

When more than one tape volume is required to contain a file or files, the tapes constitute a volume set. VAX/VMS processes volume sets as follows:

- When a file is being created on a tape volume, and the tape reaches its end-of-volume, the magnetic tape control program (ACP) sends a message to a designated system operator requesting the operator to mount another tape volume. The program that is attempting to write to the tape must wait until the operator (or user who is performing operator functions) responds to the request. The response generally includes the initialization of another tape, with a volume number one greater than the volume number of the current volume.
- When a file that spans two or more volumes is being read and the tape reaches end-of-tape, the magnetic tape ACP sends a message to the system operator requesting the operator to mount the next tape in the volume set.

Normally, RMS requests new volumes automatically. However, a PL/I program can request that the next volume in a volume set be mounted, for either an input or an output operation, by calling the NEXT_VOLUME built-in subroutine (described in Section 16.4.4).

The physical process of volume switching, whether the switching is performed automatically by RMS or as a result of a call to the NEXT_VOLUME built-in subroutine, is transparent to the PL/I program. As a user, you may wish to function as an operator to receive the volume switching requests and to mount the volumes yourself. For a description of the procedure for handling volume switching, see the *VAX/VMS Command Language User's Guide*.

18.3 Relative Files

This section describes the organization of a relative file, suggests considerations for creating and using relative files, and shows examples of some typical relative file I/O operations.

18.3.1 The Organization of a Relative File

Relative organization is suitable for files with data that can be arranged serially and uniquely identified by an integer value, for example, a part number or an employee identification number. Within the file, records are written into numbered cells. There is a one-to-one correspondence between the cell number and the integer value associated with the data in the record. This number, called the relative record number, is the key by which records are written and accessed.

Figure 18–1 illustrates a relative file in which not all cells contain records. The first record written to the file was relative record number 1 (which may have been data for a part numbered 1 or for an employee whose identification number was 1, for example). The second record written was relative record number 2. The third was relative record number 4; thus cell number three does not contain a record.

Although the cells in a relative file have the same length, the records need not be fixed-length records. However, when a record is smaller than the length of a cell, the unused space is wasted.

18.3.2 Creating a Relative File

In VAX–11 PL/I, relative organization is the default for files opened with the KEYED attribute. Thus, when a WRITE statement is directed to a file with the KEYED and OUTPUT attributes, VAX–11 PL/I creates a relative file.

When you create a relative file in a PL/I program, you can specify the following ENVIRONMENT options:

- MAXIMUM_RECORD_NUMBER
- MAXIMUM_RECORD_SIZE

Considerations for specifying values for them in PL/I are given in Sections 18.3.2.1 and 18.3.2.2, respectively.

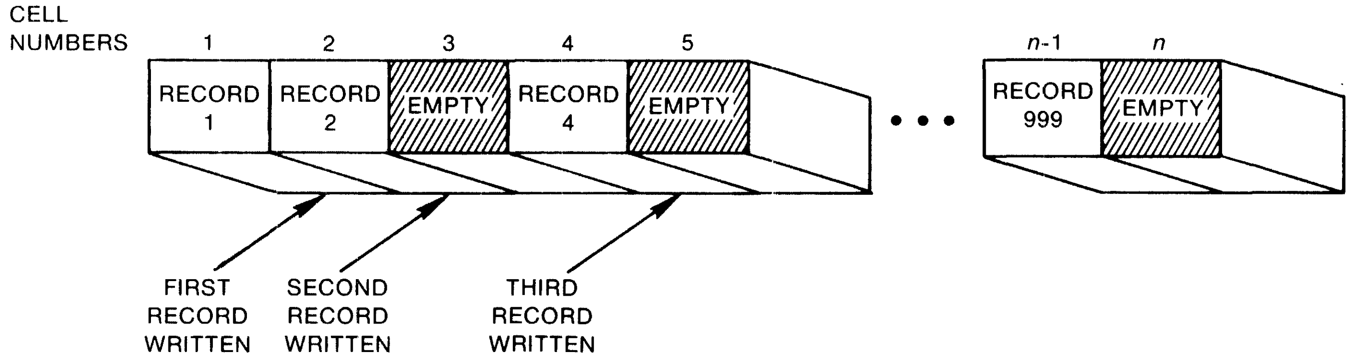


Figure 18-1: A Relative File

18.3.2.1 Maximum Record Number

The MAXIMUM_RECORD_NUMBER option specifies the largest relative record number that will be used in the file, and thus the maximum number of cells that the file can have. If you do not specify it, VAX-11 PL/I sets the maximum record number to zero, thus permitting the file to expand to any size.

For example, if a relative file is to contain inventory data about 600 parts, and the part number is to be used as the relative file's key, the MAXIMUM_RECORD_NUMBER option can be specified as follows:

```
DECLARE PARTS FILE ENVIRONMENT (
            MAXIMUM_RECORD_NUMBER (600));
```

You should be realistic and allow for future expansion of the file when you specify a maximum record number; the number is a permanent attribute of the file and cannot be changed. PL/I signals the KEY condition when a key value is too large. For an example of a PL/I ON-unit for a KEY condition, see Section 16.5.2.

18.3.2.2 Maximum Record Size

When you specify the length of the records in a file, RMS uses the value you specify in the MAXIMUM_RECORD_SIZE option to calculate a cell size. It uses the following formulas to calculate the size:

Fixed-Length Records

cell-size = 1 + record-size

One byte is required for overhead: this byte contains a deletion indicator.

Variable-Length Records

cell-size = 3 + maximum-record-size

Three bytes are required for overhead: two bytes for the individual record size and one byte for a deletion indicator.

When you select a record size for a relative file, you should try to specify a size no greater than the largest record that will be written. Otherwise, any unused space in each cell will be wasted. If you do not specify a maximum record size for either fixed- or variable-length records, VAX-11 PL/I uses the default length of 480 bytes.

18.3.3 Using Relative Files

You can create a relative file from any existing file that is suitable for such organization. The following program illustrates copying a sequential file with fixed-length records into a relative file. The circled numbers refer to the notes below.

```
COPY_TO_RELATIVE: PROCEDURE OPTIONS(MAIN);

%INCLUDE PARTLIST;  /* declaration of PARTLIST */
DECLARE 1 PARTLIST,                              ❶
          2 NAME CHARACTER(20) VARYING,
          2 NUMBER CHAR(3),
          2 UNIT_PRICE FIXED DECIMAL (5,2),
          2 QUANTITY,
            3 IN_STOCK FIXED(31),
            3 ON_ORDER FIXED(31);
                                    /* end of INCLUDE */

DECLARE OLDFILE FILE INPUT RECORD SEQUENTIAL;
ON ENDFILE(OLDFILE) STOP;                       ❷

DECLARE PARTS FILE OUTPUT KEYED RECORD ENVIRONMENT(
                MAXIMUM_RECORD_NUMBER(600),      ❸
                FIXED_LENGTH_RECORDS,
                MAXIMUM_RECORD_SIZE(38));

DECLARE RECORD_NUMBER FIXED BIN(15);

        OPEN FILE(OLDFILE);
        OPEN FILE(PARTS);                       ❹
LOOP:   READ FILE(OLDFILE) INTO(PARTLIST);
        RECORD_NUMBER = PARTLIST.NUMBER;
        WRITE FILE(PARTS) FROM(PARTLIST)
              KEYFROM(RECORD_NUMBER);           ❺
        GOTO LOOP;
END;
```

❶ The structure PARTLIST describes the layout of the records in the file. They will be ordered in the relative file according to part number, that is, using the field PARTLIST.NUMBER.

❷ The file OLDFILE is the sequential file containing the records to be copied. When the end-of-file is reached, the STOP statement terminates the program.

❸ The relative file PARTS is declared with a maximum record number of 600. It has fixed-length, 38-byte records.

❹ As each record is read into the structure PARTLIST, the value of NUMBER is copied to the fixed binary integer RECORD_NUMBER. The part number is maintained in each record in its character-string form.

❺ Each WRITE statement copies the record to the output file, specifying the value of the part number as a relative record number.

Records in this file can subsequently be accessed either sequentially or by part number. To access a record by part number, you specify the number as a key. For example:

```
GET LIST(INPUT_NUMBER) OPTIONS(PROMPT('Part? '));
READ FILE(PARTS) INTO(PARTLIST) KEY(INPUT_NUMBER);
```

Here, the value entered in response to the GET statement is used as a key value to access a record in the file.

18.3.3.1 Updating a Relative File

To add or modify records in a relative file, open it with the DIRECT and UPDATE attributes. For example, a procedure that updates the file PARTS when new stock is ordered might contain the following:

```
ORDER_PARTS: PROCEDURE (ORDERED_AMOUNT,PART_NUMBER);
%INCLUDE PARTLIST;       /* Declaration of PARTLIST */
DECLARE (ORDERED_AMOUNT,PART_NUMBER) FIXED BIN(15);
DECLARE PARTS FILE RECORD DIRECT UPDATE;

       OPEN FILE(PARTS);
       READ FILE(PARTS) INTO(PARTLIST)
                        KEY(PART_NUMBER);
       PARTLIST.ON_ORDER = PARTLIST.ON_ORDER +
                           ORDERED_AMOUNT;
       REWRITE FILE(PARTS) FROM(PARTLIST);
       CLOSE FILE(PARTS);
END;
```

The procedure ORDER_PARTS receives as its parameters the order quantity and the part number. It reads the record associated with the part number from the file, adds the order quantity to the existing quantity, and rewrites the record.

18.3.3.2 Reading a Relative File Sequentially

You can access a relative file sequentially as well as by key. In that case, each READ statement returns the record in the next cell that contains a record, skipping empty cells. The following example illustrates reading a relative file sequentially:

```
PRINT_PART: PROCEDURE OPTIONS(MAIN);
%INCLUDE PARTLIST;       /* Declaration of PARTLIST */

DECLARE PARTS FILE,
        CHECK_NUM FIXED;

        OPEN FILE(PARTS) INPUT SEQUENTIAL RECORD KEYED;
        ON ENDFILE(PARTS) STOP;
```

```
LOOP:
        READ FILE(PARTS) INTO(PARTLIST) KEYTO(CHECK_NUM);

        PUT SKIP EDIT(PARTLIST.NAME,      /* Output data */
                     UNIT_PRICE,
                     IN_STOCK,
                     ON_ORDER)
                (A,X,A,X,A,X,A);
        PUT SKIP EDIT('Relative record number',CHECK_NUM,
              'Part number:',PARTLIST.NUMBER)
            (X(10),A,X,F(5),A,X,A);       /* Output
                                             verification */

        GOTO LOOP;

END;
```

This procedure outputs the contents of the file PARTS, listing each field in the data records described by PARTLIST. The READ statement specifies the KEYTO option; the procedure outputs the value returned to the variable CHECK_NUM.

18.3.3.3 Error Handling

PL/I signals the KEY condition when errors occur while processing record numbers for relative files. For example, it signals the KEY condition when a relative record number exceeds the maximum record number specified for the file, or when the number of a record that already exists is specified in a KEYFROM option in a WRITE statement.

Section 16.5.2 contains an example of an error handler designed to handle the KEY condition.

18.4 Indexed Sequential Files

This section describes the organization of indexed sequential files, suggests considerations for creating and using them, and shows examples of some typical operations on them.

18.4.1 Indexed File Organization

In an indexed sequential file, the file contains data records and pointers to the records. Data records and record pointers are arranged in buckets, which consist of an integral number of physically contiguous 512-byte disk blocks.

Individual records within the file are located by specifying the keys associated with the records. Each file must have a primary key: that is a field within the record that has a unique value to distinguish it from all other records in the file. An indexed sequential file can also have up to 254 alternate keys, which need not have unique values.

As RMS writes records to an indexed file, it writes them in collating sequence according to the primary key, in buckets that are chained together. Thus, the file can be accessed sequentially using any key.

Figure 18–2 illustrates an indexed sequential file with a single key, or index. The records in this file consist of address data that might have been defined in a PL/I structure as follows:

```
DECLARE 1 ADDRESS_FILE,
          2 EMPLOYEE_NAME CHARACTER(30),
          2 ADDRESS,
            3 STREET CHARACTER(20),
            3 ZIP_CODE CHARACTER(5);
```

The key here is the employee name.

When RMS writes records to an indexed sequential file, it builds and maintains a tree-like structure of key value and location pointers. When records are accessed by key, RMS uses the tree to locate individual records. Thus, when a PL/I program wants to access the record whose key value is JONES, RMS traverses the indexes to locate the record.

When new records are added to an indexed sequential file, a data bucket may not have enough room to accommodate a new record. In this case, RMS performs what is called bucket splitting—it inserts a new bucket in the chain of data buckets and moves enough records from the previous bucket to preserve the primary key sequence. Bucket splitting is transparent to the PL/I program; the program only knows that it has added a record to the file.

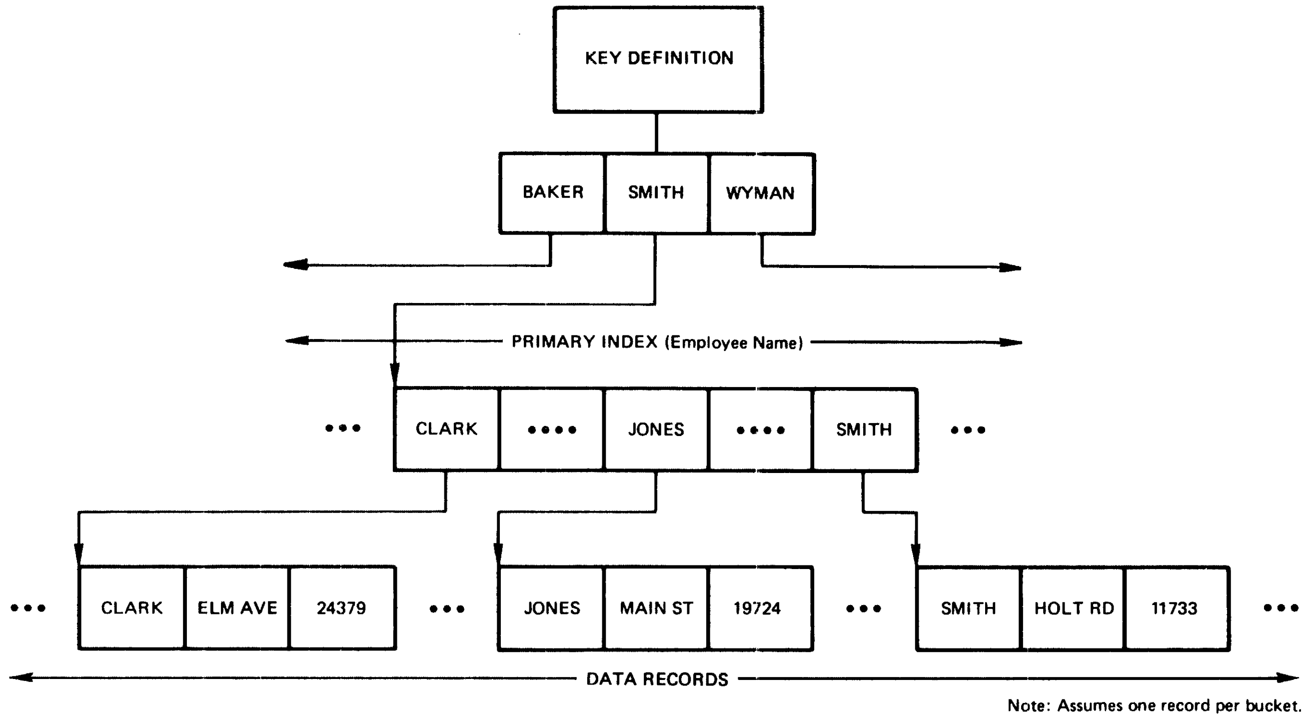


Figure 18-2: An Indexed Sequential File

18.4.2 Defining an Indexed Sequential File

To create an indexed sequential file for VAX-11 PL/I, you can use the RMS utility program EDIT/FDL. After you create the file, you can use PL/I to populate the file by opening it with the UPDATE attribute and using WRITE statements to write records to it.

To invoke the EDIT/FDL Utility, enter the following command:

```
$ EDIT/FDL
```

This utility is interactive: it prompts you to enter data and responds with error messages when you enter data incorrectly. It also provides information when you enter a question mark (?) in response to any of its prompts. The only information that you must specify is

- The file specification of the file you are creating.
- The file type IDX, to indicate that the file is an indexed sequential file.
- The position of the key within the file's records.
- The size of the key.

You can obtain help by pressing RET in response to the first menu. Each item on the menu leads to another menu in the creation of a file. You can return to the original menu by pressing CTRL/Z. Default values are not supplied. The arrows (<–) indicate lines where you enter information.

```
$EDIT/FDL(RET)
_File:ADDRESS.DAT(RET)


                         Parsing Definition File
%FDL-I-OPENFDL, error opening DBA0:[SMITH]ADDRESS.DAT;
-RMS-E-FNF, file not found
        A new FDL file will be created.
        Press return to continue.RET


                     - VAX-11 FDL Editor

    Add       to insert one or more lines into the FDL definition
    Delete    to remove one or more lines from the FDL definition
    Exit      to leave the FDL Editor after creating the FDL file
    Help      to obtain information about the FDL Editor
    Invoke    to initiate a script of related questions
    Modify    to change existing line(s) in the FDL definition
    Quit      to abort the FDL Editor with no FDL file creation
    View      to display the current FDL definition

    Main Editor Function          (Keyword)[Help]:ADD RET<-
```

```
                    - Legal Primary Attributes -

   TITLE     is the header line for the FDL file
   SYSTEM    is useful for cross-system compatibility
   FILE      attributes affect the entire RMS data file
   DATE      attributes set the date parameters of the file
   RECORD    attributes set the non-key aspects of each record
   AREA x    attributes define the characteristics of file area x
   KEY y     attributes define the characteristics of key y

   Enter Desired Primary         (Keyword)[-]    :KEY 0(RET)<-
```

At this point in the dialog, EDIT/FDL prompts you to enter key definition information.

After you define your data file, EDIT/FDL displays the message:

```
        Created:
        DBA0:[SMITH]ADDRESS.DAT;1
$
```

Table 18–2 summarizes the valid data types for keys in VAX–11 RMS indexed sequential files, lists the corresponding PL/I data type declaration, and shows how to specify the key data type and length to the EDIT/FDL utility.

18.4.3 Using Indexed Sequential Files

After you have created an indexed sequential file with the EDIT/FDL Utility, you can write records to it by opening it with the UPDATE attribute and using PL/I WRITE statements. For example:

```
OPEN FILE(STATE_FILE) RECORD DIRECT UPDATE;
    .
    .
    .
WRITE FILE(STATE_FILE) FROM(STATE)
                       KEYFROM(STATE.NAME);
```

This WRITE statement writes the record whose key value is specified by the field STATE.NAME in the structure STATE.

When a WRITE statement adds a record to an indexed sequential file, the value of the KEYFROM option must always be the primary key. In fact, the WRITE statement causes the index number (that is, the number of the key) to be reset to zero (that is, back to the primary key) if any other index number is in effect.

Table 18-2: Key Data Types

Data Type	PL/I Declaration	EDIT/FDL Specifications
String[1]	CHAR(n), where $1 \le n \le 255$	STR n
15-bit signed integer	FIXED BINARY(15)	INT 2
31-bit signed integer	FIXED BINARY(31)	INT 4
16-bit unsigned binary[2]	FIXED BINARY(15)	BIN 2
32-bit unsigned binary[2]	FIXED BINARY(31)	BIN 4
Packed decimal	FIXED DECIMAL(n) where $1 \le n \le 16$	PAC n

1. VAX-11 PL/I supports segmented character-string keys.
2. PL/I does not distinguish between signed and unsigned integers. Thus, the difference between signed integer keys and unsigned binary keys is in the key collating sequence. For signed integer keys, the collating sequence is from the smallest negative number to the largest positive number (for example -32768, -32767, ... 0, 1, 2, ... 32767). For unsigned binary keys, the collating sequence is from 0 to the largest positive number, then from the smallest negative number to -1 (for example 0, 1, 2, ... 32766, 32767, -32768, -32767, ... -1).

18.4.3.1 Reading an Indexed Sequential File Sequentially

To read records in an indexed sequential file in collating order by key value, open the file with the INPUT and SEQUENTIAL attributes. The following example illustrates reading the file STATE_FILE in sequential order using the primary key, that is, using the STATE.NAME field. This procedure uses the SET option of the READ statement; thus, no space is required in the procedure for an input buffer for the records.

```
DECLARE STATE_PTR POINTER,
        STATE_FILE FILE,
        EOF BIT(1) INITIAL ('0'B);
DECLARE 1 STATE BASED (STATE_PTR),
          2 NAME CHARACTER(20),
             .
             .
             .
```

```
ON ENDFILE(STATE_FILE) EOF = '1'B;

        OPEN FILE(STATE_FILE) INPUT SEQUENTIAL;
        READ FILE(STATE_FILE) SET(STATE_PTR);
        DO WHILE (^EOF);
            PUT SKIP(2) LIST('State:',STATE.NAME);
            PUT SKIP(2) EDIT('Population:',STATE.POPULATION)
                         (A,P'ZZ,ZZZ,ZZZ');
                .
                .
                .
            READ FILE(STATE_FILE) SET(STATE_PTR);
            END;
```

18.4.3.2 Accessing Records by Alternate Key

You can define alternate keys for an indexed sequential file when you create the file with the EDIT/FDL Utility. For complete information on alternate keys, see the *VAX-11 PL/I User's Guide*.

To read a record in an indexed sequential file using an alternate key, specify the INDEX_NUMBER option on a READ statement. For example, if a file containing data about states has as its primary key the state name, it might have alternate keys for state flowers, birds, and so on. Assuming that a field called FLOWER is the first alternate key, you could access the record for a state whose flower is MAGNOLIA by writing the following statements:

```
OPEN FILE(STATE_FILE) KEYED INPUT;
READ FILE(STATE_FILE) SET(STATE_PTR) KEY('MAGNOLIA')
     OPTIONS(INDEX_NUMBER(1));
```

The INDEX_NUMBER option specifies the first alternate index, the FLOWER field. This option is also valid on the REWRITE and DELETE statements.

You can access a file starting with an alternate index by opening the file with the INDEX_NUMBER option of ENVIRONMENT. For example:

```
OPEN FILE(STATE_FILE) SEQUENTIAL INPUT ENV(
           INDEX_NUMBER(2));
READ FILE(STATE_FILE) SET(STATE_PTR);
DO WHILE (^EOF);
    PUT SKIP EDIT(STATE.BIRD,'is the bird of',
                  STATE.NAME)
                 (A,X,A,X,A);
    READ FILE(STATE_FILE) SET(STATE_PTR);
    END;
```

These statements, executed until the end-of-file is reached, access the records in the file STATE_FILE based on its second alternate index, the BIRD field.

18.4.3.3 Updating Records in an Indexed Sequential File

You can modify or delete records in an indexed sequential file by opening it with the UPDATE attribute and using REWRITE and DELETE statements. The following example shows the correction of an invalid field in a record in the file STATE_FILE:

```
DECLARE (STATENAME,NEWFLOWER) CHARACTER(30) VARYING;
        .
        .
        .
    OPEN FILE(STATE_FILE) KEYED SEQUENTIAL UPDATE;
    GET SKIP LIST(STATENAME)
                 OPTIONS (PROMPT('State: '));
    READ FILE(STATE_FILE) SET(STATE_PTR)
             KEY(STATENAME);
    GET SKIP LIST(NEWFLOWER) OPTIONS(
                 PROMPT('New state flower name: '));
    STATE.FLOWER = NEWFLOWER;
    REWRITE FILE(STATE_FILE);
```

The REWRITE statement rewrites the current record in the file, that is, the one just read with the READ SET statement.

18.4.3.4 Error Handling

PL/I signals the KEY condition when errors occur during processing of keys for indexed sequential files. For example, if a key value specified on a READ statement indicates a key that does not exist, or a WRITE statement attempts to write a record using a primary key value that already exists, the KEY condition is signaled.

Section 16.5.2 contains an example of an error handler designed to handle the KEY condition.

Chapter 19

Built-In Functions

This chapter describes all the VAX–11 PL/I built-in functions. Section 19.1 discusses them in general, and provides a functional summary. Section 19.2 contains the individual descriptions of the built-in functions in alphabetic order.

Several of the built-in functions have corresponding pseudovariables that can be used on the left side of an assignment statement instead of the right side. See Section 12.5 for a description of these pseudovariables.

19.1 Summary of Built-In Functions

Built-in functions are procedures provided by the PL/I language, which can be used wherever an expression is valid. The built-in functions are summarized in Table 19–1, according to the following functional categories:

- Arithmetic built-in functions—provide information about the properties of arithmetic values, or perform common arithmetic calculations
- Mathematical built-in functions—perform standard mathematical calculations in floating point
- String-handling built-in functions—process character-string and bit-string values
- Conversion built-in functions—convert data from one data type to another
- Condition-handling built-in functions—provide information about interrupts caused by signaled conditions
- Array-handling built-in functions—provide information about arrays
- Storage control built-in functions—return values concerning based variables
- Timekeeping built-in functions—return the system date and time of day

- File control built-in functions—return the current line number and page number of a file
- Miscellaneous—check the validity of data, and aid in argument passing

Table 19-1: Summary of PL/I Built-In Functions

Category	Function Reference	Value Returned
Arithmetic	ABS(x)	Absolute value of x
	ADD(x,y,p[,q])	Value of x+y, with precision p and scale factor q
	CEIL(x)	Smallest integer greater than or equal to x
	DIVIDE(x,y,p[,q])	Value of x divided by y, with precision p and scale factor q
	FLOOR(x)	Largest integer that is less than or equal to x
	MAX(x1,x2)	Larger of the values x1 and x2
	MIN(x1,x2)	Smaller of the values x1 and x2
	MOD(x,y)	Value of x modulo y
	MULTILPY(x,y,p[,q])	Value of x*y, with precision p and scale factor q
	ROUND(x,k)	Value of x rounded to k digits
	SIGN(x)	-1, 0, or 1 to indicate the sign of x
	TRUNC(x)	Integer portion of x
Mathematical	ACOS(x)	Arc cosine of x (the angle, in radians, whose cosine is x)
	ASIN(x)	Arc sine of x (the angle, in radians, whose sine is x)
	ATAN(y[,x])	If x is omitted, the arc tangent of y (the angle, in radians, whose tangent is y); if x is supplied, the arc tangent of y/x (the angle, in radians, whose tangent is y/x)
	ATAND(y[,x])	If x is omitted, the arc tangent of y (the angle, in degrees, whose tangent is y); if x is supplied, the arc tangent of y/x (the angle, in degrees, whose tangent is y/x)
	ATANH(x)	Hyperbolic arc tangent of x
	COS(x)	Cosine of radian angle x
	COSD(x)	Cosine of degree angle x
	COSH(x)	Hyperbolic cosine of x
	EXP(x)	Base of the natural logarithm, *e*, to the power x
	LOG(x)	Logarithm of x to the base *e*
	LOG10(x)	Logarithm of x to the base 10
	LOG2(x)	Logarithm of x to the base 2
	SIN(x)	Sine of the radian angle x
	SIND(x)	Sine of the degree angle x
	SINH(x)	Hyperbolic sine of x
	SQRT(x)	Square root of x

Table 19-1 (Cont.): Summary of PL/I Built-In Functions

Category	Function Reference	Value Returned
Mathematical	TAN(x)	Tangent of the radian angle x
	TAND(x)	Tangent of the degree angle x
	TANH(x)	Hyperbolic tangent of x
String-Handling	BOOL(x,y,z)	Result of Boolean operation z performed on x and y
	COLLATE( )	ASCII character set
	COPY(s,c)	c copies of specified string, s
	HIGH(c)	String of length c of repeated occurrences of the highest character in the collating sequence
	INDEX(s,c)	Position of the character string c within the string s
	LENGTH(s)	Number of characters or bits in the string s
	LOW(c)	String of length c of repeated occurrences of the lowest character in the collating sequence
	SEARCH(s,c)	Position of the first character in s that is found in c
	STRING(s)	Concatenation of values in array or structure s
	SUBSTR(s,i[,j])	Part of string s beginning at i for j characters
	TRANSLATE(s,c[,d])	String s with substitutions defined in c and d
	TRIM(s[,e,f])	String s with all characters in e removed from the left, and all characters in f removed from the right
	VERIFY(s,c)	Position of the first character in s that is not found in c
Conversion	BINARY(x[,p[,q]])	Binary value of x with precision p and scale factor q
	BIT(s[,l])	Value of s converted to a bit string of length l
	BYTE(x)	ASCII character represented by the integer x
	CHARACTER(s[,l])	Value of s converted to a character string of length l
	DECIMAL(x[,p[,q]])	Decimal value of x
	FIXED(x,p[,q])	Fixed arithmetic value of x
	FLOAT(x,p)	Floating arithmetic value of x
	RANK(c)	Integer representation of the ASCII character c
	UNSPEC(x)	Internal coded form of x
Condition-Handling	ONARGSLIST( )	Pointer to argument lists of exception condition
	ONCODE( )	Error code of the most recent run-time error
	ONFILE( )	Name of file constant for which the most recent ENDFILE, ENDPAGE, KEY, or UNDEFINEDFILE condition was signaled
	ONKEY( )	Value of key that caused KEY condition

Table 19-1 (Cont.): Summary of PL/I Built-In Functions

Category	Function Reference	Value Returned
Array-Handling	DIMENSION(x,n)	Extent of the nth dimension of x
	HBOUND(x,n)	Higher bound of the nth dimension of x
	LBOUND(x,n)	Lower bound of the nth dimension of x
Storage	ADDR(x)	Pointer identifying the storage referenced by x
	ALLOCATION(x)	Number of existing generations for controlled variable x
	INT(x[,p[,l]])	Signed integer value of variable x, located at position p with length l
	NULL( )	A null pointer value
	OFFSET(p,a)	An offset into the location in area a pointed to by pointer p
	POINTER(o,a)	A pointer to the location at offset o within area a
	POSINT(x[,p[,l]])	Unsigned integer value of variable x, located at position p with length l
	SIZE(x)	Number of bytes allocated to variable x
Timekeeping	DATE( )	System date in the form yymmdd
	TIME( )	System time of day in the form hhmmssxx
File Control	LINENO(x)	Line number of the print file identified by x
	PAGENO(x)	Page number of the print file identified by x
Miscellaneous	DESCRIPTOR(x)	Forces its argument to be passed by descriptor to a non-PL/I procedure
	VALID(p)	Boolean value, indicating whether the pictured variable p has a value consistent with its picture specification

Built-in functions are similar to operators, and their arguments, to operands. Note the following restrictions on built-in function arguments:

- All arguments of all built-in functions except the array-handling, storage, file control, and STRING functions must be scalars of arithmetic, string, or pictured data types, as specified for the individual function.
- A reference to a built-in function that takes no arguments must still contain the pair of enclosing parentheses [example: NULL()] unless the function's name has been declared with the BUILTIN attribute.

Built-in functions, like other operations, can signal conditions. The mathematical functions, which are computed in floating point, can signal OVERFLOW and UNDERFLOW under the appropriate conditions. Functions that are computed in fixed point can signal FIXEDOVERFLOW. In general, string and other functions signal ERROR if a result cannot be computed.

19.2 Built-In Function Descriptions

This section contains the individual descriptions of the built-in functions in alphabetic order.

ABS Built-In Function

The ABS built-in function returns the absolute value of an arithmetic expression x. Its format is

```
ABS(x)
```

For example:

```
A = 3.567;
Y = ABS(A);                                /*   Y = +3.567    */

A = -3.567;
Y = ABS(A);                                /*   Y = +3.567    */

ROOT = SQRT (ABS(TEMP));
```

The last example shows a common use for the ABS built-in function, that is, to ensure that an expression has a positive value before requesting the square root (SQRT) built-in function is requested.

ACOS Built-In Function

The ACOS built-in function returns a floating-point value that is the arc (inverse) cosine of an arithmetic expression x. The arc cosine is computed in floating point. The returned value is an angle w such that

$$0 \leq w \leq \pi$$

The absolute value of x, after its conversion to floating point, must be less than or equal to 1. The format of the function is

```
ACOS(x)
```

ADD Built-In Function

The ADD built-in function returns the sum of two arithmetic expressions x and y, with a specified precision p and an optionally specified scale factor q.

The format of the function is

```
ADD(x,y,p[,q])
```

p

An unsigned integer constant greater than zero and less than or equal to the maximum precision of the result type (31 for fixed-point data, 34 for floating-point decimal data, and 113 for floating-point binary data).

q

An integer constant less than or equal to the specified precision. The scale factor may be optionally signed when used in fixed-point binary addition. The scale factor for fixed-point binary must be in the range -31 to p. The scale factor for fixed-point decimal data must be in the range 0 to p. If you omit q, the default value is zero. A scale factor is not to be used for floating-point arithmetic.

Expressions x and y are converted to their derived type before the addition is performed. See Appendix A.

ADDR Built-In Function

The ADDR built-in function returns a pointer to storage denoted by a specified variable. The only restriction on the variable reference is that it be addressable. The format of the function is

```
ADDR(reference)
```

If the reference is to a parameter (or any element or member of a parameter), the pointer value obtained must not be used after return from the parameter's procedure invocation (for example, by saving the pointer in a static variable or returning it as a function value).

See Section 9.5.1.1 for a general discussion of pointer values.

ASIN Built-In Function

The ASIN built-in function returns a floating-point value that is the arc (inverse) sine of an arithmetic expression x. The arc sine is computed in floating point. The returned value is an angle w such that

$$-\pi/2 \leq w \leq \pi/2$$

The absolute value of x, after its conversion to floating point, must be less than or equal to 1. The format of the function is

```
ASIN(x)
```

ALLOCATION Built-In Function

The ALLOCATION built-in function returns a fixed-point binary integer that is the number of extant generations of a specified controlled variable. If no generations of the specified variable exist, the function returns 0. The format of the function is

```
{ ALLOCATION }
{ ALLOCN     } (reference)
```

reference

The name of a controlled variable.

The following example illustrates one of the uses of the ALLOCATION built-in function:

```
DECLARE INPUT CHARACTER(10) CONTROLLED,
A CHARACTER(3) VARYING;
        .
        .
        .
    DO UNTIL (INPUT = 'QUIT');
       ALLOCATE INPUT;
       GET LIST(INPUT);
        .
        .
        .
    END;
    A = ALLOCATION(INPUT);
    PUT SKIP LIST('Generations = 'A);
```

This example uses the ALLOCATION built-in function to return the number of generations of the controlled variable INPUT. The example illustrates how input in an interactive program can be stored on a stack for future use.

ATAN Built-In Function

The ATAN built-in function returns a floating-point value that is the arc tangent of an arithmetic expression y or an arc tangent computed from two arithmetic expressions y and x. The arc tangent is computed in floating point. If two arguments are supplied, they must not both be zero after their conversion to floating point.

The format of the function is

```
ATAN(y[,x])
```

The returned value represents an angle in radians.

If x is omitted, the returned value v equals arc tangent(s), such that

$$-\pi/2 < v < \pi/2$$

where s is the value of expression y after its conversion to floating point.

If x is present, the returned value v equals arc tangent(s/r), such that

if $s \geq 0$ then $0 \leq v \leq \pi$, and
if $s < 0$ then $-\pi < v < 0$

where s and r are, respectively, the values of expressions y and x after their conversion to floating point.

ATAND Built-In Function

The ATAND built-in function returns a floating-point value that is the arc tangent of a single arithmetic expression y or an arc tangent computed

from two arithmetic expressions y and x. The arc tangent is computed in floating point. If two arguments are supplied, they must not both be zero after their conversion to floating point.

The format of the function is

```
ATAND(y[,x])
```

The floating-point value returned, representing an angle in degrees, equals ATAN(y,x)*180/π.

ATANH Built-In Function

The ATANH built-in function returns a floating-point value that is the inverse hyperbolic tangent of an arithmetic expression x. After its conversion to floating point, the absolute value of the argument x must be less than 1.

The format of the function is

```
ATANH(x)
```

BINARY Built-In Function

The BINARY built-in function converts an arithmetic or string expression x to its binary representation, with an optionally specified precision p and scale factor q. The returned binary value is either fixed or floating point, depending on whether x is a fixed- or floating-point expression.

The precision p, if specified, must be an integer constant greater than zero and less than or equal to the maximum precision of the result type (31 if fixed-point binary and 113 if floating-point binary). P must be specified if x is a fixed-point decimal value with fractional digits.

The scale factor q, if specified, must be an integer constant in the range –31 to 31 and must be less than or equal to p.

The format of the function is

```
{ BINARY }
{ BIN    } (x[,p[,q]])
```

The result type is fixed- or floating-point binary, depending on whether the argument x is a fixed- or floating-point expression. (If the argument is a bit- or character-string expression, the result type is fixed-point binary.)

The argument x is converted to the result type, giving a value v, following the usual rules for conversion (see Appendix A for details).

The returned value is the value v, with precision p. If p is omitted (integer and floating-point arguments only), the precision of the returned value is

the converted precision of x. FIXEDOVERFLOW, OVERFLOW, or UNDERFLOW is signaled if appropriate.

BIT Built-In Function

The BIT built-in function converts an arithmetic or string expression x to a bit string of an optionally specified length. If x is a string expression, it must consist of 0s and 1s. If the length is specified, it must be a nonnegative integer. If the length is omitted, the returned value has a length determined by the usual rules for conversion to bit strings (see Appendix A).

The format of the function is

```
BIT(x[,length])
```

BOOL Built-In Function

The BOOL built-in function performs a Boolean operation on two bit-string arguments and returns the result as a bit string with the length of the longer argument. Its format is

```
BOOL(string-1,string-2,operation-string)
```

string-1

A bit-string expression of any length.

string-2

A bit-string expression of any length.

operation-string

A bit-string expression that is converted to length 4. Each bit in the operation string specifies the result of comparing two corresponding bits in string-1 and string-2. Specify bit positions in the operation string from left to right to define the operation, as follows:

string-1-bit	string-2-bit	Result Specified as
0	0	Bit 1 of operation string
0	1	Bit 2 of operation string
1	0	Bit 3 of operation string
1	1	Bit 4 of operation string

If string-1 and string-2 are of different lengths, the smaller is extended on the right with zeros to the length of the larger.

For example:

```
X = '101010'B;
Y = '110011'B;
CHECK = BOOL (X,Y,'0110'B);
```

The operation is the exclusive OR. The result is '011001'B. Figure 19–1 illustrates this example.

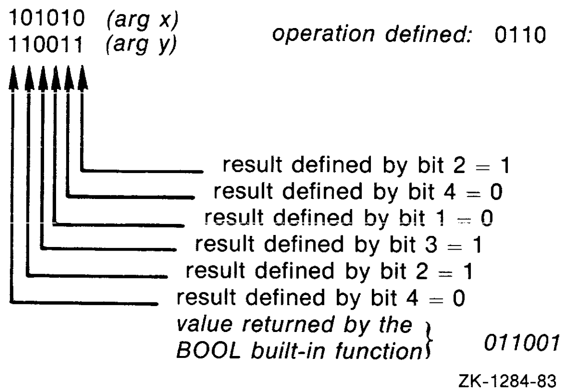


FIGURE 19-1: Example of the BOOL Built-In Function

BYTE Built-In Function

The BYTE built-in function returns the ASCII character whose ASCII code is the integer x; x must not be negative. The returned value is a character equivalent to BYTE(y), where y equals x modulo 128. The format of the function is

```
BYTE(x)
```

For example:

```
DECLARE CHAR CHARACTER(1);

CHAR = BYTE(65);         /* CHAR = 'A' */
CHAR = BYTE(32);         /* CHAR = ' ' (space) */
```

CEIL Built-In Function

The CEIL function returns the smallest integer that is greater than or equal to an arithmetic expression x. Its format is

```
CEIL(x)
```

If x is a floating-point expression, a floating-point value is returned with the same precision as x. If x is a fixed-point expression, the returned value is a fixed-point value of the same base as x and with

precision = min(31,p−q+1)
scale factor = 0

where p and q are the precision and scale factor of x.

For example:

```
A = 4.3;
Y = CEIL(A);     /* Y = 5 */

A = -4.3;
Y = CEIL(A);     /* Y = -4 */
```

CHARACTER Built-In Function

The CHARACTER built-in function converts an arithmetic or string expression x to a character string of an optionally specified length. If the length is specified, it must be a nonnegative integer. If the length is omitted, the length of the returned value is determined by the usual rules for conversion to character strings (see Appendix A). The format of the function is

```
CHARACTER(x[,length])
```

For example:

```
CHAR:       PROCEDURE OPTIONS(MAIN);

DECLARE EXPRES FIXED DECIMAL(7,5);
DECLARE OUTPUT PRINT FILE;

EXPRES = 12.34567;

OPEN FILE(OUTPUT) TITLE('CHAR2.OUT');

PUT SKIP FILE(OUTPUT)
            LIST('No length argument: ',
                 CHARACTER(EXPRES));

PUT SKIP FILE(OUTPUT)
            LIST('Length = 4: ',
                 CHARACTER(EXPRES,4));

END CHAR;
```

The program CHAR produces the following output:

```
No length argument:       12.34567
Length = 4:        12
```

In the first PUT LIST statement, CHARACTER has only one argument, so the entire string is written out. The string '12.34567' is actually preceded by two spaces; such is the case with any nonnegative number converted to a character string (see Appendix A). In the second PUT LIST statement, CHARACTER has a length argument of 4, so the first four characters of the converted string are written out: ' 12'.

COLLATE Built-In Function

The COLLATE built-in function returns a 256-character string consisting of the ASCII character set in ascending order. Its format is

```
COLLATE()
```

COPY Built-In Function

The COPY built-in function copies a given string a specified number of times and concatenates the result into a single string. Its format is

```
COPY(string,count)
```

string

Any bit- or character-string expression. If the expression is a bit string, so is the result. Otherwise, the result is a character string.

count

Any expression that yields a nonnegative integer. The specified count controls the number of copies of the string that are concatenated, as follows:

Value of Count	String Returned
0	a null string
1	the string argument
n	n concatenated copies of the string argument

For example, the function reference

```
COPY('12',3)
```

returns the character-string value '121212'.

COS Built-In Function

The COS function returns a floating-point value that is the cosine of an arithmetic expression x, where x represents an angle in radians. The cosine is computed in floating point. The format of the function is

```
COS(x)
```

COSD Built-In Function

The COSD built-in function returns a floating-point value that is the cosine of an arithmetic expression x, where x is an angle in degrees. The cosine is computed in floating point. The format of the function is

```
COSD(x)
```

COSH Built-In Function

The COSH built-in function returns a floating-point value that is the hyperbolic cosine of an arithmetic expression x. The hyperbolic cosine is computed in floating point. The format of the function is

```
COSH(x)
```

DATE Built-In Function

The DATE built-in function returns a 6-character string in the form yymmdd, where

yy is the current year (00–99)
mm is the current month (01–12)
dd is the current day of the month (01–31)

Its format is

```
DATE()
```

DECIMAL Built-In Function

The DECIMAL built-in function converts an arithmetic or string expression x to a decimal value of an optionally specified precision p and scale factor q.

P and q, if specified, must be integer constants. P must be greater than zero and less than or equal to the maximum precision for the result type (31 for fixed-point, 34 for floating-point decimal). If q is specified, x must be a fixed-point expression and p must also be specified; if q is omitted, the scale factor of the result is zero.

The format of the function is

```
{ DECIMAL }
{ DEC     }  (x[,p[,q]])
```

The result type is fixed-point or floating-point decimal, depending on whether x is a fixed- or floating-point expression. (If x is a bit- or character-string expression, the result type is fixed-point decimal.)

The expression x is converted to a value v of the result type, following the usual rules (see Appendix A for details). The returned value is v with precision p and scale factor q. If p and q are omitted, they are the converted precision and scale factor of x. FIXEDOVERFLOW, UNDERFLOW, or OVERFLOW is signaled if appropriate.

DESCRIPTOR Built-In Function

The DESCRIPTOR built-in function forces its argument to be passed by descriptor to a non-PL/I procedure. The corresponding parameter descriptor must specify the ANY attribute without the VALUE attribute. A reference to the built-in function may occur only as an argument in such a context and has no other use. The format of the function is

```
DESCRIPTOR(expression)
```

expression

The argument to be passed by descriptor. Its data type must be computational but may not be pictured. (It may be an array variable.)

DIMENSION Built-In Function

The DIMENSION built-in function returns a fixed-point binary integer that is the number of elements in a specified dimension of an array. Its format is

```
{ DIMENSION }
{ DIM       }  (reference,dimension)
```

reference

The name of an array variable.

dimension

An integer constant specifying the dimension of the array for which the extent is to be determined.

For example:

```
INIT: PROCEDURE (ARRAY);
DECLARE ARRAY(*) FIXED,
        I FIXED;

        DO I = 1 TO DIM(ARRAY,1);
            ARRAY(I) = I;
            END;
```

This procedure is passed a one-dimensional array of an unknown extent. The DIMENSION built-in function is used as the end value in a controlled DO statement. This DO-group assigns integral values to each element of the array ARRAY so that the first element has the value 1, the second element has the value 2, and so on to the last element of the array.

DIVIDE Built-In Function

The DIVIDE built-in function divides an arithmetic expression x by an arithmetic expression y and returns the quotient with a specified precision p and optionally specified scale factor q. The scale factor q must be an integer following these rules:

- If either x or y is fixed binary, q must be between –31 and 31.
- If both x and y are fixed decimal, q must not be negative.
- If either x or y is floating point, q must be zero.
- If q is omitted, it is assumed to be zero.

The expressions x and y are converted to their derived types before the division is performed (see Section 12.4.1). If y is zero after this conversion, the ZERODIVIDE condition is signaled. The quotient has the derived type of the two arguments.

The format of the function is

```
DIVIDE(x,y,p[,q])
```

EXP Built-In Function

The EXP built-in function returns a floating-point value that is the base *e* to the power of an arithmetic expression x. The computation is performed in floating point. The format of the function is

```
EXP(x)
```

FIXED Built-In Function

The FIXED built-in function converts an arithmetic or string expression x to a fixed-point arithmetic value with a specified precision p and, optionally, a scale factor q.

The format of the function is

```
FIXED(x,p[,q])
```

p

The number of bits used to represent the arithmetic value. The precision must be greater than zero and less than or equal to 31.

q

An integer in the range 0 to 31 for decimal data, and in the range -31 to 31 for binary data. If q is omitted, it is assumed to be zero. The scale factor q must be less than or equal to the specified precision.

The result type is fixed-point binary or decimal, depending on whether x is binary or decimal. (If x is a bit-string expression, the result type is fixed-point binary; if x is a character-string expression, the result type is fixed-point decimal.)

The expression x is converted to a value v of the result type, following the usual rules (see Appendix A for details). The returned value is v with precision p and scale factor q. If q is omitted, the returned value has the converted precision of x, and a scale factor of zero. FIXEDOVERFLOW is signaled if appropriate.

FLOAT Built-In Function

The FLOAT built-in function converts a string or arithmetic expression x to floating point, with a specified precision. P must be an integer constant that is greater than zero and less than or equal to the maximum precision of the result type (34 for floating-point decimal, 113 for floating-point binary).

The format of the function is

```
FLOAT(x,p)
```

The result type is floating-point binary or decimal, depending on whether x is a binary or decimal expression. (If x is a bit-string expression, the result type is floating-point binary; if x is a character-string expression, the result type is floating-point decimal.)

The expression x is converted to a value v of the result type, following the usual rules (see Appendix A). The value returned is v to the specified precision; UNDERFLOW or OVERFLOW is signaled if appropriate.

FLOOR Built-In Function

The FLOOR built-in function returns the largest integer that is less than or equal to an arithmetic expression x. Its format is

```
FLOOR(x)
```

If x is a floating-point expression, the returned value is floating point. If x is a fixed-point expression, the returned value is fixed point with the same base as x and with the attributes

precision = min(31,p–q+1)
scale factor = 0

where p and q are the precision and scale factor of x.

For example:

```
A = 3;
Y = FLOOR(A);                          /* Y = 3.00 */

A = -3.32;
Y = FLOOR(A);                          /* Y = -4.00 */
```

HBOUND Built-In Function

The HBOUND built-in function returns a fixed-point binary integer that is the upper bound of a specified dimension of an array. Its format is

```
HBOUND(reference,dimension)
```

reference

The name of an array variable.

dimension

An integer constant indicating a dimension of the specified array.

HIGH Built-In Function

The HIGH built-in function returns a string of specified length that consists of repeated appearances of the highest character in the collating sequence. Its format is

```
HIGH(length)
```

length

The specified length of the returned string. The maximum length of the returned string is 32767 characters.

The rank of the highest character that can appear in the collating sequence for VAX–11 PL/I is ASCII 255.

INDEX Built-In Function

The INDEX built-in function returns a fixed-point binary integer that indicates the position of a specified substring within a string. The value

returned indicates the position of the leftmost occurrence of the substring. If it is not found, or if the length of either argument is zero, the INDEX function returns zero.

The format of the function is

INDEX(string,substring)

string

The string to be searched for the given substring. It can be either a character- or bit-string expression.

substring

The substring to be located. It must have the same string data type as the string argument.

For example:

```
DECLARE RESULT FIXED BINARY(31),
        NEW_STRING CHARACTER(80);
RESULT = INDEX('ABCDEF','DEF');
        /* RESULT equals 4
        (DEF begins at fourth position) */
RESULT = INDEX('SHARP FORTUNE','R');
        /* RESULT equals 4
        (leftmost occurrence of R at fourth position) */
NEW_STRING = '315-54-3159';
IF INDEX(NEW_STRING,'-') = 4 THEN
        GO TO SOCIAL_SECURITY;
        /* Expression is TRUE */
```

INT Built-In Function

The INT built-in function treats specified storage as a signed integer, and returns the value of the integer. Its format is

INT(expression[,position[,length]])

expression

A scalar expression or reference to connected storage. If position and length are not specified, the length of the referenced storage must not exceed 32 bits.

position

A positive integer value that denotes the position of the first bit in the field. If omitted, position defaults to 1, signifying the first bit of storage denoted by the expression. If specified, position must satisfy the condition

$1 \leq \text{position} \leq \text{size(expression)}$

where size(expression) is the length in bits of the storage denoted by expression. A position equal to size(expression) implies a zero-length field.

length

An integer value in the range 0 to 32 that specifies the length of the field. If omitted, length is the number of bits from the bit denoted by position through the end of the storage denoted by expression. If specified, length must satisfy the condition

$0 \leq \text{length} \leq \text{size(expression)} - \text{position}$

where size(expression) is the length in bits of the storage denoted by expression.

The value returned by INT is of the type FIXED BINARY (31). If the field has a length of zero, INT returns zero.

The following example shows the INT built-in function as used to interpret the storage occupied by a bit string as an integer:

```
B16 = '0000000000001101'B;      /* 16-bit string */

I = BIN(B16);                   /* I = 13 */
I = INT(B16);                   /* I = -20480 */

B64 = '5076ABCD00000000'B4;     /* 64-bit string */

I = INT(B64,1,32);               /* First 32 bits; I = -1277858294 */
I = INT(B64,33);                 /* Second 32 bits; I = 0 */
I = INT(B64);                    /* Field too large,
                                    run-time error */
```

Notice that, unlike the BIN built-in function, the INT built-in function performs no conversion. It simply treats the contents of the designated storage as a signed integer. Therefore, the value returned by INT depends on the data type (and therefore the internal representation) of the variable occupying the storage.

LBOUND Built-In Function

The LBOUND built-in function returns a fixed-point binary integer that is the lower bound of a specified dimension of an array. Its format is

LBOUND(reference,dimension)

reference

The name of the array variable.

dimension

An integer constant indicating the dimension of the specified array.

LENGTH Built-In Function

The LENGTH built-in function returns a fixed-point binary integer that is the number of characters or bits in a specified character- or bit-string

expression. For a varying-length character string, the function returns its current length. The format of the function is

```
LENGTH(string)
```

LINENO Built-In Function

The LINENO built-in function returns a FIXED BINARY(15) integer that is the current line number of the referenced print file. Its format is

```
LINENO(reference)
```

If the file is closed, the returned value is the last value from the previous opening. If the file was never opened, the returned value is zero.

LOG Built-In Function

The LOG built-in function returns a floating-point value that is the base *e* (natural) logarithm of an arithmetic expression x. The computation is performed in floating point. The expression x must be greater than zero after its conversion to floating point.

The format of the function is

```
LOG(x)
```

LOG10 Built-In Function

The LOG10 built-in function returns a floating-point value that is the base 10 logarithm of arithmetic expression x. The computation is performed in floating point. The expression x must be greater than zero after its conversion to floating point. The format of the function is

```
LOG10(x)
```

LOG2 Built-In Function

The LOG2 built-in function returns a floating-point value that is the base 2 logarithm of an arithmetic expression x. The computation is performed in floating point. The expression x must be greater than zero after its conversion to floating point. The format of the function is

```
LOG2(x)
```

LOW Built-In Function

The LOW built-in function returns a string of specified length that consists of repeated appearances of the lowest character in the collating sequence. Its format is

```
LOW(length)
```

length

The specified length of the returned string. The maximum length permitted is 32767 characters.

The rank of the lowest character that can appear in the collating sequence for VAX-11 PL/I is ASCII 0.

MAX Built-In Function

The MAX built-in function returns the larger of two arithmetic expressions x and y. The format of the function is

```
MAX(x,y)
```

The expressions x and y are converted to their derived type (see Section 12.4.1) before the operation is performed. If the derived type is floating point, the value returned is also floating point, with the larger precision of the two converted arguments. If the derived type is fixed point, the returned value is a fixed-point value with the base of the derived type and with the attributes

precision = min(31,max(px–qx,py–qy)+max(qx,qy))
scale factor = max(qx,qy)

where px,qx and py,qy are the converted precisions and scale factors of x and y, respectively.

MIN Built-In Function

The MIN built-in function returns the smaller of two arithmetic expressions x and y. Its format is

```
MIN(x,y)
```

The expressions x and y are converted to their derived type (see Section 12.4.1) before the operation is performed. If the derived type is floating point, the value returned is also floating point, with the larger precision of the two converted arguments. If the derived type is fixed point, the returned value is fixed point, with the base of the derived type and with the attributes

precision = min(31,max(px–qx,py–qy)+max(qx,qy))
scale factor = max(qx,qy)

where px,qx and py,qy are the converted precisions and scale factors of x and y.

MOD Built-In Function

The MOD built-in function returns, for an arithmetic expression x and nonnegative arithmetic expression y, the value r that equals x modulo y. That is, r is the smallest positive value that must be subtracted from x to make the remainder exactly divisible by y. (The result when y is negative is explained below.)

The format of the function is

```
MOD(x,y)
```

The expressions x and y are converted to their derived type (see Section 12.4.1) before the operation is performed. If the derived type is fixed-point binary or unscaled fixed-point decimal, then the result precision is the precision of the second operand. If the derived type is floating point, the returned value is an approximation in floating point, with the larger precision of the two converted arguments.

The value returned is

u-w*floor(u/w)

where u and w are the arguments x and y, respectively, after conversion to their derived type. If w is zero, u is converted to the precision described below, which may signal FIXEDOVERFLOW.

If x and y are fixed-point expressions, the returned value is fixed point with the attributes

precision = min(31,pw-qw+max(qu,qw))
scale factor = max(qu,qw)

where qu is the scale factor of u, pw is the precision of w, and qw is the scale factor of w. The FIXEDOVERFLOW condition is signaled if

pw-qw + max(qu,qw) > 31

For example:

```
MODEX: PROCEDURE OPTIONS(MAIN);

DECLARE OUTMOD PRINT FILE;

ON FIXEDOVERFLOW PUT FILE(OUTMOD)
        SKIP LIST('FIXEDOVERFLOW signaled');

PUT FILE(OUTMOD) SKIP LIST(MOD(28,128));
PUT FILE(OUTMOD) SKIP LIST(MOD(130,128));
PUT FILE(OUTMOD) SKIP LIST(MOD(-28,128));
PUT FILE(OUTMOD) SKIP LIST(MOD(4.5,.758));
PUT FILE(OUTMOD) SKIP LIST(MOD(-4.5,.758));
PUT FILE(OUTMOD) SKIP LIST(MOD(1.5E-3,-1.4E-3));
PUT FILE(OUTMOD) SKIP LIST(MOD(28,0));

END MODEX;
```

The program MODEX writes the following output to OUTMOD.DAT:

```
    28
     2
   100
 0.710
 0.048
-1.3E-03

FIXEDOVERFLOW signaled      8
```

The last PUT statement attempts to evaluate MOD(28,0). The constants 28 and 0 are both fixed-point decimal expressions, with precisions (2,0) and (1,0), respectively. Therefore, the attributes of the returned value are determined to be FIXED DECIMAL, with

precision = min(31,1–0+max(0,0)) = 1
scale factor = max(0,0) = 0

Although 28 modulo 0 is 28, MOD(28,0) signals FIXEDOVERFLOW because 28 cannot be represented in the result precision. (The value of the function is therefore undefined.)

MULTIPLY Built-In Function

The MULTIPLY built-in function multiplies two arithmetic expressions x and y, and returns the product of the two values with a specified precision p and an optionally specified scale factor q.

The format of the function is

```
MULTIPLY(x,y,p[,q])
```

p

An integer constant greater than zero and less than or equal to the maximum precision of the result type (31 for fixed-point binary data, 34 for floating-point decimal data, and 113 for floating-point binary data.)

q

An integer in the range –31 to p when used with fixed-point binary multiplication. The scale factor for fixed-point decimal multiplication has a range 0 to p. A scale factor is not to be used with floating-point arithmetic. If no scale factor is designated, q defaults to zero.

Expressions x and y are converted to their derived type before the multiplication is performed.

NULL Built-In Function

The NULL built-in function returns a null pointer value. Its format is

```
NULL()
```

For example:

```
IF NEXT_POINTER = NULL() THEN CALL FINISH;
```

The IF statement checks whether the pointer variable NEXT_POINTER is null; if so, it executes the CALL statement.

OFFSET Built-In Function

The OFFSET built-in function converts a pointer to an offset relative to a

designated area. If the pointer is null, the result is null. The format of the function is

```
OFFSET(pointer,area)
```

pointer

A reference to a pointer variable whose current value either represents the location of a based variable within the specified area or is null.

area

A reference to a variable declared with the AREA attribute. If the specified pointer is not null, it must designate a storage location within this area.

For example:

```
DECLARE MAP_SPACE AREA (2048),
        START OFFSET (MAP_SPACE),
        QUEUE_HEAD POINTER;
    .
    .
    .
START = OFFSET (QUEUE_HEAD,MAP_SPACE);
```

The offset variable START is associated with the area MAP_SPACE. The OFFSET built-in function converts the value of the pointer to an offset value.

ONARGSLIST Built-in Function

The ONARGSLIST built-in function returns a pointer to the location in memory of the argument list for an exception condition. If the ONARGSLIST built-in function is referenced in any context outside of an ON-unit, it returns a null pointer. Its format is

```
ONARGSLIST()
```

The format of the argument list and the information it makes available to an ON-unit are described in the *VAX-11 PL/I User's Guide*.

ONCODE Built-In Function

The ONCODE built-in function returns as a fixed-point binary integer the status value of the most recent run-time error that signaled the current ON condition. The function may be used in any ON-unit to determine which specific error caused the condition. If the function is used in any context other than an ON-unit, it returns zero. Its format is

```
ONCODE()
```

For details on the condition values returned by ONCODE and examples of using the ONCODE built-in function, see Sections 15.1.5 and 16.5.2.

ONFILE Built-In Function

The ONFILE built-in function returns the name of the file constant for which the current file-related condition was signaled. Its format is

```
ONFILE()
```

This built-in function can be used in any of the following ON-units:

- An ON-unit established for the KEY, ENDFILE, ENDPAGE, and UNDEFINEDFILE conditions
- A VAXCONDITION ON-unit established for I/O errors that can occur during file processing
- An ERROR ON-unit that receives control as a result of the default PL/I action for file-related errors, which is to signal the ERROR condition

The returned value is a varying-length character string. If referenced outside an ON-unit or within an ON-unit that is executed as a result of a SIGNAL statement, the ONFILE function returns a null string.

ONKEY Built-In Function

The ONKEY built-in function returns the key value that caused the KEY condition to be signaled during an I/O operation to a file that is being accessed by key. Its format is

```
ONKEY()
```

This built-in function can be used in an ON-unit established for these conditions:

- The KEY, ENDFILE, or UNDEFINEDFILE conditions
- An ERROR ON-unit that receives control as a result of the default PL/I action for the KEY condition, which is to signal the ERROR condition

The returned key value is a varying-length character string. If referenced outside an ON-unit, or within an ON-unit executed as a result of the SIGNAL statement, the ONKEY built-in function returns a null string.

PAGENO Built-In Function

The PAGENO built-in function returns a FIXED BINARY(15) integer that is the current page number in the referenced print file. The print file must be open. The format of the function is

```
PAGENO(reference)
```

POINTER Built-In Function

The POINTER built-in function returns a pointer to the location identified by the referenced offset and area. Its format is

POINTER(offset,area)

offset

A reference to a variable whose current value either represents the offset of a based variable within the specified area or is null.

area

A reference to a variable that is declared with the AREA attribute and with which the specified offset value is associated.

The returned value is of type POINTER. If the offset value is null, the result is null.

For example:

```
DECLARE MAP_SPACE AREA (2048),
        START OFFSET (MAP_SPACE),
        P POINTER;
            .
            .
            .
P = POINTER (START,MAP_SPACE);
```

The POINTER built-in function converts the value of the offset variable START in the area MAP_SPACE to a pointer value.

POSINT Built-In Function

The POSINT built-in function treats specified storage as an unsigned integer, and returns the value of the integer. Its format is

POSINT(expression[,position[,length]])

expression

A scalar expression or a reference to connected storage. If position and length are not specified, the length of the referenced storage must not exceed 32 bits.

position

A positive integer value that denotes the position of the first bit in the field. If omitted, position defaults to 1, thus signifying the first bit of the storage denoted by expression. If specified, position must satisfy the condition

1 ≤ position ≤ size(expression)

where size(expression) is the length in bits of the storage denoted by expression. A position equal to size(expression) implies a zero-length field.

length

An integer value in the range 0 to 32 that specifies the length of the field. If omitted, length is the number of bits from the bit denoted by position through the end of the storage denoted by expression. If specified, length must satisfy the condition

$0 \leq$ length $\leq$ size(expression) − position

where size(expression) is the length in bits of the storage denoted by expression.

The value returned by POSINT is of the type FIXED BINARY (31). If the field has a length of zero, POSINT returns zero.

Since the POSINT built-in function treats storage as if it contained an unsigned integer, the value returned may be larger than the maximum positive value that can be contained in the signed integer that is stored in the same number of bits. Specifically, if the argument to POSINT is 32 bits long and has the high-order (sign) bit set, the resulting value is too large for assignment to a FIXED BIN (31) variable, the largest integer available in PL/I. The results of such an operation are undefined.

The use of the POSINT built-in function is identical to the use of the INT built-in function with the exception that POSINT treats its argument as an unsigned integer. The example that follows illustrates this difference.

```
DECLARE (X15,Y15,I15,P15) FIXED BIN (15),
      P31 FIXED BIN (31);

    X15 = 585;
    Y15 = -585;
    I15 = INT(X15);     /* I15 = 585 */
    I15 = INT(Y15);     /* I15 = -585 */
    P15 = POSINT(X15); /* P15 = 585 */
    P31 = POSINT(Y15); /* P31 = 64951 */
    P15 = POSINT(Y15); /* ERROR signaled */
```

In this example, POSINT first assigns the storage referenced by X15 to P15. Since this storage is occupied by a positive integer and therefore has the sign bit clear, POSINT behaves exactly like INT. However, when POSINT is applied to storage occupied by a negative integer, it interprets the set sign bit as representing part of the integer. When the resulting value is assigned to a FIXED BIN (31) variable, the value is larger than the largest possible FIXED BIN (15) value (32767). An attempt to assign the same value to a FIXED BIN (15) variable results in PL/I signaling ERROR.

RANK Built-In Function

The RANK built-in function returns a fixed-point binary integer that is the ASCII code for the designated character. The precision of the returned value is 15. The format of the function is

```
RANK(character)
```

character

Any expression yielding a 1-character value.

For example:

```
CODE = RANK('A'); /* CODE = 65 */
CODE = RANK('a'); /* CODE = 97 */
CODE = RANK('$'); /* CODE = 36 */
```

Appendix C contains a list of the ASCII characters and their corresponding numeric codes.

ROUND Built-In Function

The ROUND built-in function rounds a fixed-point decimal expression to a specified number of decimal places. Its format is

```
ROUND(expression,position)
```

expression

An arithmetic expression that yields a fixed-point decimal value with a nonzero scale factor, or a pictured value with fractional digits.

position

A nonnegative integer constant specifying the number of decimal places in the rounded result.

If the arguments are an expression of type FIXED DECIMAL(p,q) and position k, the returned value is the rounded value with the attributes

precision = max(1,min(p–q+k+1,31))
scale factor: k

The rounded value is

$$\text{ROUND}(x,k) = \text{sign}(x) * (10^{-k}) * \text{floor}(\ \text{abs}(x) * (10^{k}) + .5\ )$$

For example:

```
A = 1234.567;
Y = ROUND(A,1)
          /*    Y = 1234.6    */

Y = ROUND(A,0);                              /*    Y = 1235    */

A = -1234.567;
Y = ROUND(A,2)
          /*    Y = -1234.57   */
```

SEARCH Built-In Function

The SEARCH built-in function examines two strings and returns the second string position of the first appearance of any character in the first string that also occurs in the second string. Its format is

```
SEARCH(string-1,string-2)
```

string-1

A character-string expression to be located in the second string.

string-2

A character-string expression to be searched by the first string.

The returned value is a positive integer representing the position in string-1 of the first character in string-1 that was found in string-2. If no match is found, the returned value is zero.

For example:

```
A: PROCEDURE OPTIONS (MAIN;
DECLARE STR CHARACTER(13),
        ST2 CHARACTER(10) INITIAL ('ABCDEFGHIJ'),
        X FIXED DECIMAL (2);

STR = 'FIND';
X = SEARCH (ST2,STR);

/* X=4  D was found in the fourth position */


STR = 'ABSCAM';
X = SEARCH (STR,ST2);

/* X=1  A was found first, even though other
characters appear in both strings */


STR = 'JEEPERS!';
X = SEARCH (ST2,STR);

/* X=5  E was the first letter found */
```

SIGN Built-In Function

The SIGN built-in function returns 1, -1, or 0, indicating whether an arithmetic expression is positive, negative, or zero. The returned value is a

fixed-point binary integer. The format of the function is

```
SIGN(expression)
```

SIN Built-In Function

The SIN built-in function returns a floating-point value that is the sine of an arithmetic expression x, where x is an angle in radians. The sine is computed in floating point. The format of the function is

```
SIN(x)
```

SIND Built-In Function

The SIND built-in function returns a floating-point value that is the sine of an arithmetic expression x, where x represents an angle in degrees. The sine is computed in floating point. The format of the function is

```
SIND(x)
```

SINH Built-In Function

The SINH built-in function returns a floating-point value that is the hyperbolic sine of an arithmetic expression x. The hyperbolic sine is computed in floating point. The format of the function is

```
SINH(x)
```

SIZE Built-In Function

The SIZE built-in function returns a fixed-point binary integer that is the number of bytes allocated to a referenced variable. Its format is

```
SIZE (reference)
```

reference

The name of a variable known to this block. The reference can be to a scalar variable, an array or structure, or a structure member. The reference cannot be to a constant or expression. Although references to individual array elements are allowed, the returned value in this instance is the size of the entire array, not the element.

The returned value is the variable's allocated size in bytes. In the case of bit strings that do not exactly fill an integral number of bytes, the value is rounded up to the next byte.

For varying character-string variables, note that the returned value is two bytes greater than the declared length of the string. These extra two bytes are allocated by PL/I to contain the current length of the string.

The following example illustrates the use of the SIZE built-in function on some scalar variables:

```
DECLARE S FIXED BINARY(31),
        INT FIXED BINARY(15),
        CHAR1 CHARACTER(5),
        CHAR2 CHARACTER(5) VARYING,
        BITSTRING BIT(10),
        P POINTER;

    S = SIZE(INT);            /* S = 2 */
    S = SIZE(CHAR1);          /* S = 5 */
    S = SIZE(CHAR2);          /* S = 7 */
    S = SIZE(BITSTRING);      /* S = 2 */
    S = SIZE(P);              /* S = 4 */
```

Note the difference between the allocated size for the fixed-length and varying character strings. Note also that the returned value for the bit string is rounded up to 2 bytes, the integral number of bytes required to contain 10 bits.

SQRT Built-In Function

The SQRT built-in function returns a floating-point value that is the square root of an arithmetic expression x. The square root is computed in floating point. After its conversion to floating point, x must be greater than or equal to zero.

The format of the function is

```
SQRT(x)
```

STRING Built-In Function

The STRING built-in function concatenates the elements of an array or structure and returns the result. Elements of a string array are concatenated in row-major order. Members of a structure are concatenated in the order in which they were declared.

The format of the STRING built-in function is

```
STRING(reference)
```

reference

A reference to a variable that is suitable for bit- or character-string overlay defining. Briefly, a variable is suitable if it consists entirely of characters or bits that are packed into adjacent storage locations, without gaps. (For more information, see Section 9.7.)

The string returned is of type CHARACTER or BIT, depending on whether the reference is suitable for character- or bit-string overlay defining. The length of the string is the total number of characters or bits in the base reference.

For example:

```
STRING_BIF_EXAMPLE: PROCEDURE;

DECLARE NEW_NAME CHARACTER(40);

DECLARE 1 FULL_NAME,
          2 FIRST_NAME CHARACTER(10),
          2 MIDDLE_INITIAL CHARACTER(3),
          2 LAST_NAME CHARACTER(27) ;

FIRST_NAME = 'MABEL';
MIDDLE_INITIAL = 'S.';
LAST_NAME = 'MERCER';
NEW_NAME = STRING(FULL_NAME);
         /* NEW_NAME =
   'MABEL△△△△△S.△MERCER△△△△△△△△△△△△△△△△△△△△△'
        where △ is a blank */
END STRING_BIF_EXAMPLE;
```

SUBSTR Built-In Function

The SUBSTR built-in function returns a specified substring from a string. Its format is

SUBSTR(string,position[,length])

string

A bit- or character-string expression.

position

An integer expression that indicates the position of the first bit or character in the substring. The position must be greater than or equal to one and less than or equal to LENGTH(string)+1.

length

An integer expression that indicates the length of the substring to be extracted. If not specified, length is

LENGTH(string) - position + 1

which extracts the substring beginning at the indicated position and ending at the end of the string.

The length must satisfy the condition

$0 \leq$ length $\leq$ LENGTH(string) - position + 1

The returned substring is of type BIT(length) or CHARACTER(length), depending on the type of the string argument. If the length argument is zero, the result is a null string.

For example:

```
DECLARE (NAME,LAST_NAME) CHARACTER(20),
        START FIXED BINARY(31);

NAME = 'ISAK DINESEN';
        /* NAME = 'ISAKbDINESENΔΔΔΔΔΔΔΔ'
                where Δ is a blank */

START = INDEX(NAME,' ')+1;
        /* START = 6 */

LAST_NAME = SUBSTR(NAME,START);
        /* default length = LENGTH(NAME)-START+1 =15 */
        /* LAST_NAME = 'DINESENΔΔΔΔΔΔΔΔΔΔΔΔΔ' */
```

TAN Built-In Function

The TAN built-in function returns a floating-point value that is the tangent of an arithmetic expression x, where x represents an angle in radians. The tangent is computed in floating point. After its conversion to floating point, x must not be an odd multiple of $\pi/2$.

The format of the function is

```
TAN(x)
```

TAND Built-In Function

The TAND built-in function returns a floating-point value that is the tangent of an arithmetic expression x, where x represents an angle in degrees. The tangent is computed in floating point. After its conversion to floating point, x must not be an odd multiple of 90.

The format of the function is

```
TAND(x)
```

TANH Built-In Function

The TANH built-in function returns a floating-point value that is the hyperbolic tangent of an arithmetic expression x. The hyperbolic tangent is computed in floating point. The format of the function is

```
TANH(x)
```

TIME Built-In Function

The TIME built-in function returns an 8-character string representing the current time of day in the form hhmmssxx, where

hh is the current hour (00 - 23)
mm is the minutes (00 - 59)

ss is the seconds (00 - 59)
xx is hundredths of seconds (00 - 99)

The format of the TIME built-in function is

```
TIME()
```

The time is returned as a string of type CHARACTER (8).

TRANSLATE Built-In Function

Given a character-string argument, the TRANSLATE built-in function replaces occurrences of an old character with a corresponding translation character and returns the resulting string. Its format is

```
TRANSLATE(original,translation[,oldchars])
```

original

A character-string expression in which specific characters are to be translated.

translation

A character-string expression giving replacement characters for corresponding characters in oldchars.

If the translation is shorter than oldchars, it is padded on the right with spaces to the length of oldchars before any translation occurs. If the translation is longer than oldchars, its excess characters (on the right) are ignored.

oldchars

A character-string expression indicating which characters in the original are to be replaced. If oldchars is not specified, it defaults to COLLATE().

The following steps are performed for each character (beginning at the leftmost) in the original:

1. Let original(i) be the current character in the original string, and let result(i) be the corresponding character in the resulting string.
2. Search oldchars for the leftmost occurrence of original(i).
3. If oldchars does not contain original(i), then let result(i) equal original(i). Otherwise, let j equal the position of the leftmost occurrence of original(i) in oldchars, and let result(i) equal translation(j).
4. Return to step 1.

The string returned is of type CHARACTER(length), where length is that of the original string. If the original is a null string, so is the returned value.

For example:

```
TRANSLATE_XM: PROCEDURE OPTIONS(MAIN);

DECLARE NEWSTRING CHARACTER(80) VARYING;
DECLARE TRANSLATION CHARACTER(128);
DECLARE I FIXED;
DECLARE COLLATE BUILTIN;

/* translate space to '0': */
NEWSTRING = TRANSLATE('1 2','0',' ');
PUT SKIP LIST(NEWSTRING);

/* translate letter 'F' to 'E': */
NEWSTRING = TRANSLATE('BFFLZFBUB','E','F');
PUT SKIP LIST(NEWSTRING);

/* change case of letters in sentence */
TRANSLATION = COLLATE;

DO I=66 TO 91; /* replace upper with lower */
SUBSTR(TRANSLATION,I,1) = SUBSTR(COLLATE,I+32,1);
END;
DO I=98 TO 123; /* replace lower with upper */
SUBSTR(TRANSLATION,I,1) = SUBSTR(COLLATE,I-32,1);
END;
NEWSTRING =
TRANSLATE('THE QUICK BROWN fox JUMPS OVER THE LAZY dog',
TRANSLATION);
PUT SKIP LIST(NEWSTRING);

END TRANSLATE_XM;
```

The first reference translates the string '1 2' to '102'. The second reference translates 'BFFLZFBUB' to 'BEELZEBUB'. The third reference produces the new sentence

```
'the quick brown FOX jumps over the lazy DOG'
```

TRIM Built-In Function

The TRIM built-in function accepts a character string as an argument and returns a character string that consists of the input string with specified characters removed from the left and right. TRIM takes either one or three arguments. If you supply only one argument, TRIM removes blanks from the left and right of the argument. If you supply second and third arguments, TRIM removes characters specified by those arguments from the left and right of the string, respectively.

The format of the TRIM built-in function is

```
TRIM (input-string,[beginning-chars,end-chars])
```

input-string

A character-string variable or constant. This argument supplies the string from which characters are to be trimmed.

beginning-chars

A character-string variable or constant. This argument specifies characters to be trimmed from the left of the input string. If a character that is in the first position in the input string is also present anywhere in beginning-chars, that character is removed from the input string. This process is repeated until a character is encountered on the left of the input string that is not present in beginning-chars, or until the characters in the input string are exhausted.

end-chars

A character-string variable or constant. This argument specifies characters to be trimmed from the right of the input string. The process of removing characters from the right is identical to that of removing characters from the left, except that the character in the last position is examined.

Any of the arguments to TRIM can consist of a null string; specifically, if beginning-chars or end-chars is null, no characters are removed from the corresponding end of the input string.

When only one argument is supplied, TRIM removes blanks from both ends of that argument. In other words, the following two expressions are equivalent:

```
TRIM(S)

TRIM(S,' ',' ')
```

TRUNC Built-In Function

The TRUNC built-in function changes all fractional digits in an arithmetic expression x to zeros and returns the resulting integer value. Its format is

```
TRUNC(x)
```

If x is a floating-point expression, the returned value is floating point. If x is a fixed-point expression, the returned value is fixed point with the same base as x and with the attributes

precision: min(31,p−q+1)
scale factor: 0

where p and q are the precision and scale factor of x.

UNSPEC Built-In Function

The UNSPEC built-in function returns a bit string representing the internal coded value of the referenced scalar variable, which can be of any type. The format of the function is

```
UNSPEC(reference)
```

The returned value is a bit string whose length is the number of bits occupied by the referenced variable. This length must be less than or equal

to the maximum length for bit-string data. The returned bit string contains the contents of the referenced variable's storage, the first bit in storage being the first bit in the returned value. The actual value is specific to VAX-11 PL/I and may differ from other PL/I implementations. Note that if the referenced variable is a binary integer (FIXED BINARY), the first bit in the returned value is the lowest binary digit.

For example:

```
DECLARE X CHARACTER(2), Y BIT(16);

X = 'AB';
Y = UNSPEC(X);
    .
    .
    .
DECLARE I FIXED BINARY(15);
I = 2;
PUT LIST(UNSPEC(I));
```

As a result of the first UNSPEC reference, Y contains the ASCII codes of 'A' and 'B'. The PUT LIST statement containing UNSPEC(I) prints the string

```
'0100000000000000'B
```

VALID Built-In Function

The VALID built-in function determines whether the argument x, a pictured variable, has a value that is valid with respect to its picture specification. A valid value is any of the character strings that can be created by the picture specification. The function returns '0'B if x has an invalid value, and '1'B if it has a valid value. The function can be used whenever a data item is read in with a record input (READ) statement, to ensure that the input data is valid. The format of the function is

    VALID(x)

x

A reference to a variable declared with the PICTURE attribute.

Note that pictured data is always validated (thus making the VALID function unnecessary) when it is read in with the GET EDIT statement and the P format item; the ERROR condition is signaled if the data does not conform to the picture given in the P format item. If GET LIST is used (or GET EDIT with a format item other than P), the input value is converted to conform to the pictured input target (see Appendix A for details).

For example:

```
VALP: PROCEDURE OPTIONS(MAIN);

DECLARE INCOME PICTURE '$$$$$$V.$$';
DECLARE MASTER RECORD FILE;
DECLARE I FIXED;

DO I = 1 TO 2;
READ FILE(MASTER) INTO(INCOME);
IF VALID(INCOME) THEN;
        ELSE PUT SKIP LIST('Invalid input:',INCOME);
END;

END VALP;
```

If the file MASTER.DAT contains

```
$15000.50
 50000.50
```

then the program VALP writes out

```
Invalid input:    50000.50
```

The picture '$$$$$$V.$$' specifies a fixed-point decimal number of up to seven digits, two of which are fractional. To be valid, a pictured value must consist of nine characters; the first digit must be immediately preceded by a dollar sign; the number must contain a period before the fractional digits; and each position specified by a dollar sign must contain either that sign, a digit, or a space. The second record in MASTER.DAT can be assigned by the READ statement because it has the correct size; however, the pictured value is invalid because it does not contain a dollar sign.

VERIFY Built-In Function

The VERIFY built-in function compares a string with a test string and verifies that all characters that appear in the string also appear in the test string. If not, the VERIFY built-in function returns a fixed-point binary integer that indicates the position of the first character in the string that is not present in the test string. If each character is present, the function returns the value zero.

The format of the function is

```
VERIFY(string,test-string)
```

string

A character-string expression representing the string to be verified.

test-string

A character-string expression containing the set of characters against which the string is to be verified.

For example:

```
STRING = 'HOW MUCH IS 1 PLUS 2';
ALPHABET = 'ABCDEFGHIJKLMNOPQRSTUVWXYZ ';
A = VERIFY(STRING,ALPHABET);
```

In this example, the variable ALPHABET contains the 26 uppercase letters plus the space character. The function returns a value of 13, indicating the position of the first nonalphabetic and nonspace character in STRING. Since the test string can also be a constant, you can find the first nonspace character in any string by writing

```
A = VERIFY(STRING,' ');

NEWSTRING = 'ALL LETTERS';
A = VERIFY(NEWSTRING,ALPHABET);
```

In this example, VERIFY returns a value of zero. All characters in the string NEWSTRING are present in the string ALPHABET.

Chapter 20

Compile Time Facilities

VAX-11 PL/I supports the VAX-11 Common Data Dictionary (CDD) for managing data and an embedded preprocessor for manipulation of source text at compile time. These two facilities permit you to include record definitions from a central dictionary, include application-specific text modules from text libraries, replace identifier values, control compilation flow, and generate your own diagnostic messages.

The following topics are covered in this chapter:

- The VAX-11 Common Data Dictionary
- The VAX-11 PL/I embedded preprocessor
- Using text libraries

20.1 The VAX-11 Common Data Dictionary(CDD)

The VAX-11 Common Data Dictionary (CDD) is a set of shareable data definitions, or language-independent structure declarations, which are defined by a system manager or data administrator. The CDD provides a central storage repository that can be shared and that is protected from unauthorized access. The definitions stored in the CDD help the system manager or data administrator coordinate an effective data management system.

The advantages to using the CDD are

- Record declarations are language independent.
- A single declaration helps guarantee the accuracy and reliability of data.

NOTE

The CDD is one of the many layered products offered for the VAX-11 and not all systems that use PL/I use the CDD. Therefore, PL/I CDD support is meaningful only if CDD is on your system. If you are not certain, see your system manager.

20.1.1 Creating and Maintaining a CDD

CDD data definitions are organized into a hierarchical dictionary in much the same way that files are organized in directories and subdirectories. For example, a dictionary for defining personnel data might have separate directories for each employee category (or type). Then, subdirectories for employees who are salesmen might include data definitions for records such as salary and commission history, as well as general personnel records.

CDD entries are stored as an internal form. That is, you enter descriptions of data definitions into the dictionary in a unique, general-purpose language called Common Data Dictionary Language (CDDL). Then, the CDDL compiler converts the data descriptions to an internal form, making them independent of any higher level language. When a program is compiled, CDD data definitions are drawn into higher-level language programs (provided the data attributes are consistent). Program listings include CDD data definitions in the same language as the application program.

The following examples show the same structure declaration written in both CDDL and PL/I.

Example 1:

```
PAYROLL_RECORD STRUCTURE.
  SALESMAN STRUCTURE.
    NAME                   DATATYPE IS TEXT 30.
    ADDRESS                DATATYPE IS TEXT 40.
    SALESMAN_ID            DATATYPE IS UNSIGNED NUMERIC 5.
  END SALESMAN STRUCTURE.
```

Example 2:

```
DECLARE 1 PAYROLL_RECORD,
          2 SALESMAN,
            3 NAME CHARACTER(30),
            3 ADDRESS CHARACTER(40),
            3 SALESMAN_ID PIC '(5)9';
```

The CDD provides two utilities for creating and maintaining a dictionary:

- The Dictionary Management Utility (DMU), for creating and maintaining the CDD's directory hierarchy, history lists, and access control lists.

- The Dictionary Verify/Fix Utility (CDDV), for repairing damaged dictionary files.

Once CDD records are established, they may be included and used in VAX-11 PL/I programs. At compile time, CDD records and their attributes are extracted from the designated CDD record node; then, the record's corresponding PL/I declaration is entered into the object module.

20.1.2 Using the CDD

The %DICTIONARY statement incorporates CDD data definitions into the current PL/I source file during compilation. It can occur anywhere in a PL/I source file; it need not be within a procedure. The format of the %DICTIONARY statement is

```
%DICTIONARY cdd-path;
```

cdd-path

Is any preprocessor expression. The preprocessor expression is evaluated and converted to a character string if necessary; the result is interpreted as the full or relative pathname of a CDD object. The pathname must follow all rules for forming VAX-11 CDD pathnames.

The compiler extracts the record definition from the CDD and the corresponding PL/I structure is declared in the PL/I program.

If the %DICTIONARY statement is not embedded in a PL/I language statement, that is if %DICTIONARY immediately follows a nonpreprocessor semicolon or is the first statement in the program, then the resulting structure is declared with the logical level 1 and the BASED storage attribute is furnished. The logical member levels increment from 2. For example:

```
DECLARE PRICE FIXED BINARY(31);
%DICTIONARY 'ACCOUNTS';
```

would result in a declaration of the form

```
DECLARE PRICE FIXED BINARY(31);
DECLARE 1 ACCOUNTS BASED,
          2 NUMBER,
            3 LEDGER CHARACTER(3),
            3 SUBACCOUNT CHARACTER(5),
          2 DATE CHARACTER(12),
            .
            .
            .
```

Notice that in this example, ACCOUNTS is a relative dictionary pathname.

If the %DICTIONARY statement is embedded in a PL/I language statement, as in a structure declaration, then the resulting structure is declared

with no logical level and no storage attribute. Logical member numbers are supplied and incremented from 100. For example:

```
DECLARE 1 COMMON_INTERFACES STATIC EXTERNAL,
          %DICTIONARY 'ACCOUNTS';    ,
          %DICTIONARY 'ADDRESSES';   ;
```

Notice the syntax in this example: the %DICTIONARY statement is terminated with a semicolon before the normal PL/I line terminator. In this context, the %DICTIONARY statement is always terminated with a semicolon. Other declaration punctuation must also be included. At compile time, this declaration would result in a declaration of the form

```
DECLARE 1 COMMON_INTERFACES STATIC EXTERNAL,
          100 ACCOUNTS,
            101 NUMBER,
               102 LEDGER CHARACTER(3),
               102 SUBACCOUNT CHARACTER(5),
            101 DATE CHARACTER(12),
               .
               .
               .
          100 ADDRESSES,
               .
               .
               .
```

The CDD supports data types that are not native to PL/I. PL/I makes accessible all data types by declaring unsupported data types as appropriately sized BIT_FIELD or BYTE_FIELD: the PL/I compiler does not attempt to approximate a data type that it does not support. For example, an F_FLOATING_COMPLEX number is declared BYTE_FIELD(8), not (2)FLOAT(24). If you access CDD definitions that contain data types not supported by PL/I, the compiler issues an error and supplies a BIT_FIELD or BYTE_FIELD type which gives you an opportunity to manipulate data though PL/I built-in functions such as ADDR, INT, POSINT, and UNSPEC.

PL/I issues warnings for CDD features that it does not support and issues error messages when the features conflict.

CDD data definitions can contain explanatory text in the CDDL DESCRIPTION IS clause. You can include this text in the PL/I listing comments, by specifying /SHOW=DICTIONARY. You may use these comments to indicate the data type of each structure and member. The punctuation for CDDL comments is the same as for other PL/I programs.

When you extract a record definition from the CDD, you can choose to include this translated record in the program's listing by using the /LIST/SHOW=DICTIONARY qualifiers in the PLI command line.

Even if you choose not to list the extracted record, the names, data types, and offsets of the CDD record definition are displayed in the program listing's allocation map.

20.2 The VAX-11 PL/I Embedded Preprocessor

The VAX-11 PL/I embedded preprocessor permits you to alter a source program at compile time. Preprocessor statements can be mixed with non-preprocessor statements in the source program, but preprocessor statements are executed only at compile time. The resulting source program is then used for further compilation.

The embedded preprocessor performs two types of preprocessing:

- It interprets preprocessor statements and evaluates preprocessor expressions.
- It replaces the values of preprocessor variables and procedures.

Preprocessor statements allow you to include text from alternative sources (INCLUDE libraries and the VAX-11 Common Data Dictionary), control the course of compilation (%DO, %GOTO, %PROCEDURE and %IF), issue user-generated diagnostic messages, and selectively control listings and formats. The preprocessor statements are summarized in Table 20-1 in Section 20.3.2.

20.2.1 Preprocessor Compilation Control

At compile time, preprocessor variables, procedures, and variable expressions are evaluated in the order in which they appear in the source text, and the new values are substituted in the source program in the same order. Thus, the course of compilation becomes conditional, and the resulting executable program may exhibit a variety of unique features. Note that preprocessor variables and procedures must be declared and activated before replacement occurs.

For example:

```
%DECLARE HOUR FIXED;
%HOUR = SUBSTR(TIME(),1,2);

%IF HOUR > 7 & HOUR < 18
%THEN
    %FATAL 'Please compile this outside of prime time';
%DECLARE T CHARACTER;
%ACTIVATE T NORESCAN;
%T = '''Compiled on '||DATE()||'''';
```

```
DECLARE INIT_MESSAGE CHARACTER(40) VARYING INITIAL(T);

    %IF VARIANT() = '' | VARIANT() = 'NORMAL'
    %THEN
        %INFORM 'NORMAL';
    %ELSE
        %IF VARIANT() = 'SPECIAL';
        %THEN
            %INFORM 'SPECIAL';
        %ELSE
            %IF VARIANT() = 'NONE';
            %THEN %;
            %ELSE
                %DO;
                %T = '''unknown variant''';
                %WARN T;
                INIT_MESSAGE = INIT_MESSAGE||' with '||T;
                %END;
    %END;

PUT LIST (INIT_MESSAGE);
```

This example illustrates several aspects of the embedded preprocessor. First, this program must be compiled outside of prime time. Second, depending upon the value of VARIANT, the program is compiled with a different variant.

Notice the number of single quote marks around the string constant assigned to T. Single quotes are sufficient if the value of T is used only in a preprocessor user-generated diagnostic message. That is, the value of T is concatenated with nonpreprocessor text and assigned to INIT_MESSAGE because during preprocessing, single quotes are stripped off of string constants. To ensure that the run-time program also has quotes around the string, additional quotes are needed.

20.2.2 Preprocessor Procedures

The %PROCEDURE statement defines the beginning of a preprocessor procedure block and specifies the parameters, if any, of the procedure. A preprocessor procedure executes only at compile time. Invocation is similar to a function reference and occurs in two ways:

- Preprocessor statements can invoke preprocessor procedures. In addition, preprocessor statements from within preprocessor procedures can invoke other preprocessor procedures.
- Statements from the source program can invoke preprocessor procedures.

A preprocessor procedure is invoked by the appearance of its entry name and list of arguments. If the reference occurs in a nonpreprocessor statement, the entry name must be active before the preprocessor procedure is

invoked. If the entry name is activated with the RESCAN option, the value of the preprocessor procedure is rescanned for further possible preprocessor variable replacement and procedure invocation. Preprocessor procedures may be invoked recursively.

Since the preprocessor procedure is always invoked as a function, the %PROCEDURE statement must also specify (via the RETURNS option) the data type attributes of the value that is returned to the point of invocation.

The return value replaces the preprocessor procedure reference in the invoking source code. Preprocessor procedures may not return values via their parameter list. The return value must be capable of being converted to one of the data types CHARACTER, FIXED, or BIT. The maximum precision of the value returned by the %RETURNS statement is BIT(31), CHARACTER(255), and FIXED(10).

Preprocessor procedures may have one of two distinctly different types of argument lists: positional or keyword. Positional argument lists (ending with a right parenthesis) use parameters sequentially, as in a parenthesized list. Positional argument lists may be used in any preprocessor procedure. Keyword argument lists (ending with a semicolon) use parameters in any order, as long as each keyword matches the name of a parameter. This permits the option of specifying the order in which parameters may be passed. Keyword argument lists may only be used when the preprocessor procedure contains the STATEMENT option and is invoked from a nonpreprocessor statement.

A keyword argument list ends with a semicolon rather than the right parenthesis. In this way, the STATEMENT option permits you to use a preprocessor procedure as if it were a statement. Consequently, preprocessor procedures using the STATEMENT option permit you to extend the PL/I language by simulating features that may not otherwise be available.

When using keyword arguments in nonpreprocessor statements, the keywords may be used in any order. The following reference examples would produce a variety of results with positional arguments, because values would be used sequentially. Keyword arguments produce consistent results because keyword parameters are matched with keyword arguments.

The next example illustrates the use of the STATEMENT option; to generate PL/I source statements that define a unique run time feature. The preprocessor procedure APPEND returns a string, which is incorporated into the source program at compile time. At run time, the code resulting from this PL/I source text performs the specified function.

This preprocessor procedure permits a varying string to accumulate text up to its maximum size without danger of undetected truncation. Normally, strings that exceed their maximum size are truncated. The text returned

by the preprocessor procedure provides the run-time program with a way to handle truncation. If the string would be truncated, a message is printed and the FINISH condition is signaled.

```
%APPEND: PROCEDURE (string,to) STATEMENT RETURNS(CHARACTER); ❶

    %DECLARE (string,to) CHARACTER; ❷
    %RETURN (
        'DO;'|| ❸
        'IF LENGTH('||string||')+LENGTH('||to||') > SIZE('||to||')-2'||
        ' THEN DO;'||
           'PUT SKIP LIST (''Buffer overflowed appending to '||to||''');'||
           'SIGNAL FINISH;'|| ❹
           'END;'||
        'ELSE '||to||' = '||to||'||'||string||';'||
        'END;'
             );
     %END;
```

The following notes are keyed to this example:

❶ The preprocessor procedure APPEND is defined with the parameters ´string´ and ´to´ and the STATEMENT option.

❷ ´String´ and ´to´ are declared as parameters within the preprocessor procedure.

❸ The %RETURN statement returns the value contained by the parentheses. This text then becomes part of the PL/I source program.

Notice the punctuation within the character string returned by %RETURN. At compile time, single quotes are stripped when the text is incorporated into the run-time PL/I program. In addition, the semicolon that delimits the invocation is not retained when the replacement takes place. All customary PL/I punctuation must be included in the character string.

❹ If the current varying string and the additional string together are greater than the maximum length of the varying string, an informational message is printed and the FINISH condition is signaled.

The following invocations of the preprocessor procedure APPEND are all equivalent:

```
APPEND STRING('New String') TO (My_string);
APPEND TO(My_String) STRING('New String');
APPEND('New String') TO(My_string);
```

Notice that if you have a preprocessor procedure (A) with a label that is the same as the name of a keyword argument in another preprocessor procedure (B) with the STATEMENT option, then when B is invoked the keyword argument is treated as a call to procedure A, and not as a keyword parameter in B.

For the syntax of the %PROCEDURE statement, see Section 20.3.3.19.

20.2.3 Preprocessor Statements

All statements are preceded by a percent sign (%) and are terminated by a semicolon (;). All text that appears within these delimiters is considered part of the preprocessor statement and is executed at compile time. For example:

```
%DECLARE HOUR FIXED;          /* declaration of a preprocessor
                                      single variable */

%DECLARE (A,B) CHARACTER;     /* a factored preprocessor
                                 declaration  */

%HOUR = SUBSTR(TIME(),1,2);   /* preprocessor assignment
                                  statement using two built-in
                                      functions  */
```

Notice that a percent sign (%) is required only at the beginning of the statement. Preprocessor built-in functions are contained within preprocessor statements and consequently do not require a percent sign. However, when you include Common Data Dictionary record definitions, you may need to include the usual PL/I punctuation. See Section 20.4 for details.

Labels are permitted on preprocessor statements and, like other PL/I labels, are used as the targets of program control statements. A preprocessor label must be an unsubscripted label constant and must be preceded by a percent sign. As with other preprocessor statements, the percent sign alerts the compiler that until the line is terminated with a semicolon, all subsequent text is preprocessor text. Therefore, no other percent signs are required on that line.

Labels for preprocessor procedures are necessary for the procedure to be invoked. On a preprocessor procedure, the leading percent sign is only required on the label; statements within the procedure do not require leading percent signs.

The format for a preprocessor label is

%label: preprocessor-statement;

Table 20–1 summarizes the preprocessor statements.

Table 20–1: Summary of PL/I Preprocessor Statements

Statement	Use
%Assignment	Evaluates a preprocessor expression and gives its value to a preprocessor identifier
%	Null statement, specifies no preprocessor operation
%ACTIVATE	Makes the value of declared preprocessor variables and procedures eligible for replacement
%DEACTIVATE	Makes the value of declared preprocessor variables and procedures ineligible for replacement

Table 20-1 (Cont.): Summary of PL/I Preprocessor Statements

Statement	Use
%DECLARE	Defines the preprocessor variable names and identifiers to be used in a PL/I program, and specifies the data attributes associated with them
%DICTIONARY	Specifies data definitions to be included from the VAX-11 Common Data Dictionary
%DO	Denotes the beginning of a group of preprocessor statements to be executed as a unit
%END	Denotes the end of a block or group of statements that started with a %PROCEDURE or a %DO statement
%ERROR	Generates a user-defined diagnostic error message
%FATAL	Generates a user-defined fatal diagnostic message
%GOTO	Transfers control to a labeled preprocessor statement
%IF	Tests a preprocessor expression, and establishes action to be performed based on the results
%INCLUDE	Copies the text of an external file into the source file at compile time
%INFORM	Generates a user-defined informational diagnostic message
%[NO]LIST_ALL	Does/does not include CDD records, INCLUDE files, machine code, and source statements in the listing from that point on
%[NO]LIST_DICTIONARY	Does/does not include CDD records in the listing from that point on
%[NO]LIST_INCLUDE	Does/does not include INCLUDE files in the listing from that point on
%[NO]LIST_MACHINE	Does/does not include machine code in the listing from that point on
%[NO]LIST_SOURCE	Does/does not include source program statements in the listing from that point on
%PAGE	Provides listing pagination without form feeds in the source text
%PROCEDURE	Begins a preprocessor procedure
%REPLACE	Assigns a constant value to an identifier at compile time
%RETURN	Returns a value from execution of a preprocessor procedure to the point of invocation
%SBTTL	Allows specification of a listing subtitle line
%TITLE	Allows specification of a listing title line
%WARN	Generates a user-defined warning diagnostic message

The preprocessor statements summarized in Table 20-1 are described individually in the following sections.

20.2.3.1 %Assignment Statement

The preprocessor assignment statement gives a value to a specified preprocessor variable. The format of the assignment statement is

```
%target = expression;
```

target

The name of the preprocessor variable to be assigned a value. It must be an unsubscripted reference to a preprocessor variable.

expression

Any valid PL/I expression.

For arithmetic operations, only decimal integer arithmetic of precision (10,0) is performed. Each operand and all results are converted, if necessary, to a fixed decimal value of precision (10,0). Note that fractional digits are truncated.

20.2.3.2 %Null Statement

The %NULL statement performs no action. Its format is

```
%;
```

The most common use for the preprocessor %NULL statement is as the target statement of a %THEN or %ELSE clause in an %IF statement. For example:

```
%IF
   ERROR() > 0;
%THEN
   %GOTO FIXIT;
%ELSE
   %;
```

In this example, no action is taken if the program does not generate a user-diagnostic error message. If the %GOTO does not change the flow of compilation, control passes to the next executable preprocessor statement in the source text.

20.2.3.3 %ACTIVATE Statement

The %ACTIVATE statement makes preprocessor variable and procedure identifiers eligible for replacement. If the compiler encounters the named identifier, it will initiate replacement. The format of the %ACTIVATE statement is

```
% { ACTIVATE }  element [ RESCAN   ] ,...;
  { ACT      }          [ NORESCAN ]
```

element

The name of a previously declared preprocessor identifier or a parenthesized list of identifiers that are separated by commas.

[RESCAN | NORESCAN]

Specifies if the preprocessor is or is not to continue checking the text for secondary value replacement.

The RESCAN option specifies that preprocessor scanning continue until all possible identifier replacements are completed. RESCAN is the default option.

The NORESCAN option specifies that replacement be done once only; the resulting text is not rescanned for possible further replacement.

An identifier is activated by either %ACTIVATE or %DECLARE. When an activated identifier is encountered by the compiler in unquoted nonpreprocessor statements, the variable name or procedure reference is replaced by its value. Replacement continues throughout the rest of the source program unless it is stopped with the %DEACTIVATE statement.

If an identifier which is not a preprocessor variable is the target of an %ACTIVATE statement, a warning message is issued and the identifier is implicitly declared as a preprocessor variable with the FIXED attribute. Thereafter, the identifier variable is eligible for replacement when activated.

For example:

```
DECLARE (A,B,C) FIXED;
%DECLARE (A,B) FIXED;
%ACTIVATE (A,B);

%A = 1;
%B = (A + A);

C = A + B;
PUT SKIP LIST (C);     /* C = 3 */
```

In this example, the activated preprocessor variables A and B are assigned values by the preprocessor. Notice that variables A and B are also declared as nonpreprocessor variables and are established as variables within the nonpreprocessor program. The value of C in this case is 1+2, since the preprocessor identifiers A and B are both active, and therefore replaced by their preprocessor values.

```
%DEACTIVATE B;

B = 900;
C = A + B;
PUT SKIP LIST (C);     /* C = 901 */
```

If preprocessor variable B is deactivated, then the identifier B is not replaced by its preprocessor value 2, and the value of C is 1+B.

Values used for computation vary depending on whether the preprocessor variables are activated, and therefore eligible, for replacement or whether the variables are deactivated and not eligible for replacement. The value of A is either 1 or 1100; the value for B is either 2 or 900.

Variables may be activated for replacement either with the %DECLARE or %ACTIVATE statement, but you must declare a preprocessor variable before it can be activated.

It is possible to activate several variables with a single statement. For example:

```
%DECLARE (A,B,C) FIXED;
%ACTIVATE (A,B) RESCAN, C NORESCAN;
```

RESCAN is the default action, but this example explicitly activates A and B with the RESCAN option. C is activated, but is not to be rescanned.

20.2.3.4 %DEACTIVATE Statement

The %DEACTIVATE statement makes preprocessor variable and procedure identifiers ineligible for replacement. After a variable or procedure has been deactivated, it will not be replaced in nonpreprocessor text during preprocessing. Replacement of a deactivated variable or procedure will not occur again until the identifier has been reactivated with the %ACTIVATE statement.

The format for the %DEACTIVATE statement is

```
% { DEACTIVATE }  element,...;
  { DEACT      }
```

element

The name of a previously declared preprocessor identifier or a parenthesized list of identifiers that are separated by commas and surrounded by parentheses.

For example:

```
DECLARE (A,B,C) FIXED;
%DECLARE (A,B) FIXED;
%DEACTIVATE (A);

%A = 1;
%B = (A + A);

A = 1100;
B = 900;

C = A + B;
PUT SKIP LIST (C);     /* C = 3300 */
```

In this example, the preprocessor variable A is deactivated and, therefore, the preprocessor value of 1 is not used. The preprocessor variable B is activated by default, which means that the value of B is 2200. Thus, the run-time value of variable C is the result of adding 1100 + (A + A). It is possible to deactivate several variables with a single statement. For example:

```
%DEACTIVATE (A,B,C,D,E,F);
```

deactivates six variable identifiers in a single statement.

20.2.3.5 %DECLARE Statement

The %DECLARE statement establishes an identifier as a preprocessor variable, specifies the data type of the variable, and activates the identifier for replacement. It can occur anywhere in a PL/I source program.

The format of the %DECLARE statement is

```
% { DECLARE }  element [ FIXED     ]  ,...;
  { DCL     }          [ CHARACTER ]
                       [ BIT       ]
```

element

The name of a preprocessor identifier or a parenthesized list of identifiers that are separated by commas and surrounded by parentheses. Elements must be given an attribute of BIT, FIXED, or CHARACTER, but neither precision nor length may be specified. The compiler supplies the variables with the following attributes:

Attribute	Implied Attributes
BIT	(31) INITIAL ((31)'0'B)
FIXED	DECIMAL (10,0) INITIAL (0)
CHARACTER	VARYING (255) INITIAL ('')

If no data type is specified, FIXED is assumed.

When a variable is declared in a preprocessor statement, it is activated for replacement and rescanning. The scope of a preprocessor variable is all of the text following the declaration of the variable, unless the variable is declared inside a preprocessor procedure. Using %DECLARE inside a procedure has the effect of declaring a local variable.

For example:

```
%DECLARE HOUR FIXED;
```

In this example, HOUR is declared as a preprocessor variable identifier with the FIXED attribute. The compiler supplies the default values that make this declaration the equivalent of

```
DECLARE HOUR FIXED DECIMAL (10,0) INITIAL (0);
```

NOTE

Notice that the attribute FIXED implies FIXED DECIMAL in a preprocessor declaration. In nonpreprocessor declarations, FIXED implies FIXED BINARY.

Factored declarations are permitted and follow the same usage rules as nonpreprocessor declarations. For example:

```
%DECLARE (A,B) CHARACTER, C BIT;
```

Both A and B are declared with the CHARACTER attribute. The compiler supplies default values that make this declaration the equivalent of

```
%DECLARE (A,B) CHARACTER VARYING(255) INITIAL(''),
         C BIT(31)INITIAL((31)'0'B);
```

20.2.3.6 %DICTIONARY Statement

The %DICTIONARY statement incorporates VAX–11 Common Data Dictionary (CDD) data definitions into the current PL/I source file during compilation. It can occur anywhere in a PL/I source file; it need not be within a procedure. The format of the %DICTIONARY statement is

%DICTIONARY cdd-path;

cdd-path

Is any preprocessor expression. The preprocessor expression is evaluated and converted to a CHARACTER string if necessary. The resulting character string is interpreted as the full or relative pathname of a CDD object. The resultant pathname must conform to all rules for forming VAX–11 CDD pathnames.

The compiler extracts the record definition from the CDD, and the PL/I structure corresponding to the record description is declared in the PL/I program. For example:

```
DECLARE PRICE FIXED BINARY(31);
%DICTIONARY 'ACCOUNTS';
```

would result in a declaration of the form

```
DECLARE PRICE FIXED BINARY(31);
DECLARE 1 ACCOUNTS BASED,
          2 NUMBER,
            3 LEDGER CHARACTER(3),
            3 SUBACCOUNT CHARACTER(5),
          2 DATE CHARACTER(12),
             .
             .
             .
```

Notice that in the above example, ACCOUNTS is a relative dictionary pathname.

20.2.3.7 %DO Statement

The %DO statement begins a sequence of preprocessor statements, which terminates with the %END statement. %DO statements are noniterative and must be simple DO-groups; however, preprocessor DO-groups are useful when combined with %IF statement. Preprocessor DO-groups may contain both preprocessor and nonpreprocessor text.

The format of the %DO statement is

```
%DO;
   .
   .
   .
%END;
```

For example:

```
%DECLARE T CHARACTER;      /* declare T */
%ACTIVATE T NORESCAN;      /* activate T for replacement */
     .
     .
     .
%IF VARIANT() = 'NONE';
   %THEN ;
   %ELSE
       %DO;
       %T = '''unknown variant''';  /* assign string to T */
       %WARN T;                      /* output unknown variant
                                     /warning at compile time */
       INIT_MESSAGE = INIT_MESSAGE||' with '||T; /* assign
                                     / value of T to non-
                                     / preprocessor variable */
       %END;
```

This preprocessor DO-group performs several steps. First, a string constant is assigned to T. Then, the value of T is used in a preprocessor user-generated diagnostic message. This message is issued at compile time to warn the programmer that the program is compiled with an unknown variant. Finally, the value of T is concatenated with a nonpreprocessor string constant. INIT_MESSAGE, including the value of T, is part of the run-time image.

20.2.3.8 %END Statement

The %END statement terminates a preprocessor procedure or DO-group. The format of the %END statement is

```
%END;
```

After execution of a %END statement, control passes to the next executable statement.

20.2.3.9 %ERROR Statement

The %ERROR statement provides a diagnostic error message during program compilation. The format of the %ERROR statement is

```
%ERROR preprocessor-expression;
```

preprocessor-expression

A maximum of 64 characters giving the text of the error message to be displayed. Messages of more than 64 characters are truncated.

The returned message displayed by %ERROR is

```
%PLIG-E-USERDIAG, text
```

text

The preprocessor expression specified by the %ERROR statement in the source program.

Compilation errors that result in the display of the %ERROR statement increment the informational diagnostic count displayed in the compilation summary, and inhibit production of an object file.

20.2.3.10 %FATAL Statement

The %FATAL statement provides a diagnostic fatal message during program compilation. The format of the %FATAL statement is

```
%FATAL preprocessor-expression;
```

preprocessor-expression

A maximum of 64 characters giving the text of the fatal message to be displayed. Messages of more than 64 characters are truncated.

The returned message displayed by %FATAL is

```
%PLIG-F-USERDIAG, text
```

text

The preprocessor expression specified by the %FATAL source program.

Compilation errors that result in a fatal preprocessor error terminate compilation after the message is displayed.

20.2.3.11 %GOTO Statement

The %GOTO statement causes the preprocessor to interrupt its sequential processing of source text and continue processing at the point specified in the %GOTO statement. A %GOTO is useful for avoiding large segments of text in the source program. The format of the %GOTO statement is

```
%GOTO label-reference;
```

label-reference

A label of a preprocessor statement. The label reference determines the point to which compiler processing will be transferred: only forward transfers are allowed.

The following example illustrates forward transfers and the use of %GOTO:

```
%IF WARN() = 5
   %THEN
      DO;
         .
         .
         .
         %END;
   %ELSE;
      %GOTO INSERT_TEXT;
          .
          .
          .
%INSERT_TEXT: DO;
```

Depending upon the status of the %IF statement in this example, program compilation takes one of two courses. The preprocessor DO-group is executed or control is transferred either to the statement labeled INSERT_TEXT. Notice also in this example that the preprocessor built-in function WARN is used to determine preprocessor action, which makes the program self-diagnostic.

If a %GOTO statement is used within a preprocessor procedure, the label reference must be contained within the preprocessor procedure; that is, a %GOTO must not transfer control outside of the preprocessor procedure. Likewise, a %GOTO may not transfer control into another preprocessor procedure.

20.2.3.12 %IF Statement

The %IF statement controls the flow of program compilation according to the scalar bit value of a preprocessor expression. The %IF statement tests the preprocessor expression and performs the specified action if the result of the test is true (or 1). The format of the %IF statement is

%IF test-expression %THEN action [%ELSE action]

test-expression

Any valid preprocessor expression that yields a scalar bit value. If any bit of the value is 1, then the expression is true; otherwise the expression is false.

action

A single, unlabeled preprocessor statement, %DO-group, %GOTO statement, or a preprocessor null statement. The specified action must not be an %END statement.

The %IF statement evaluates the preprocessor test expression. If the expression is true, the action specified following the keyword %THEN is compiled. Otherwise, the action, if any, following the %ELSE keyword is compiled. In either case, compilation resumes at the first statement following the termination of the %IF statement, unless a %GOTO in one of the action clauses causes compilation to resume elsewhere.

See the %DO statement (Section 20.3.2.7) for an example.

20.2.3.13 %INCLUDE Statement

The %INCLUDE statement incorporates text from other files into the current source file during compilation. It can occur anywhere in a PL/I source file, and it need not be part of a procedure. The format of the %INCLUDE statement is

```
%INCLUDE { 'file-spec'  } ;
         { module-name  }
```

file-spec

A file specification enclosed in apostrophes. The specification is subject to logical name translation and the application of default values by the VAX/VMS operating system.

module-name

The 1- to 31-character name of a text module in a library of included files and/or other text modules. If the text module is not in PLISYSDEF or has the logical name PLI$LIBRARY, the name of the library containing the module must be specified in the PLI compilation command.

For details on the specification of files and libraries to be included in a PL/I compilation, see the *VAX-11 PL/I User's Guide.*

For example:

```
%INCLUDE 'SUM.PLI';
```

This statement copies the contents of the file SUM.PLI into the current file during compilation.

```
%INCLUDE SYSTEM_PROCEDURES;
```

This statement includes a module from a text module library. The library containing the module SYSTEM_PROCEDURES must be present in the command that compiles this program.

The maximum depth to which %INCLUDE statements can be nested is four.

See Section 7.4.3 for further details.

20.2.3.14 %INFORM Statement

The %INFORM statement specifies a user-written diagnostic informational message to be displayed during program compilation. The format of the %INFORM statement is

```
%INFORM preprocessor-expression;
```

preprocessor-expression

A maximum of 64 characters giving the text of the informational message to be displayed. Messages of more than 64 characters are truncated.

The returned message displayed by %INFORM is

```
%PLIG-I-USERDIAG, text
```

text

The preprocessor expression specified by the %INFORM statement in the source program.

The %INFORM statement increments the informational diagnostic count displayed in the compilation summary.

20.2.3.15 %LIST Statement

The %LIST statement enables the selective listing display of INCLUDE file contents, extracted CDD record descriptions, machine code, and source code. The %LIST statement has a number of forms: each enables or disables listing control for specific portions of the source text. The formats of all forms of the %LIST statement are

```
%LIST_ALL;
%LIST_DICTIONARY;
%LIST_INCLUDE;
%LIST_MACHINE;
%LIST_SOURCE;
```

These statements are only effective when you give the appropriate value to the /SHOW qualifier on the PLI command.

The %LIST form of each statement enables the appearance of the specified information starting with the listing line following the %LIST statement. If you previously specified %NOLIST, the %LIST statement has the effect of reenabling the display.

The following summarizes the text displayed with each form of %LIST statement

- %LIST_ALL—displays all of the following
- %LIST_DICTIONARY—displays the PL/I translation of an included Common Data Dictionary record

- %LIST_INCLUDE—displays the contents of INCLUDE files and modules in the program listing
- %LIST_MACHINE—displays the machine language code generated during compilation
- %LIST_SOURCE—displays source program statements in the program listing

%LIST statements may not be nested.

20.2.3.16 %NOLIST Statement

The %NOLIST statement disables the selective listing display of INCLUDE file contents, extracted Common Data Dictionary (CDD) record descriptions, machine code, and source code. The %NOLIST statement has a number of forms, each of which enables or disables listing control for specific portions of the source text. The formats of all forms of the %NOLIST statement are

```
%NOLIST_ALL;
%NOLIST_DICTIONARY;
%NOLIST_INCLUDE;
%NOLIST_MACHINE;
%NOLIST_SOURCE;
```

The %NOLIST form of each statement disables the appearance of the specified information starting with the listing line following the %NOLIST statement. If you previously specified %LIST, the %NOLIST statement has the effect of disabling the display.

To cancel the effect of any of the %NOLIST statements, include %LIST at the appropriate line in the source text.

20.2.3.17 %PAGE Statement

The %PAGE statement provides listing pagination without inserting form-feed characters into the source text.

The format of %PAGE is

```
%PAGE;
```

The first source record following the record which contains the %PAGE statement is printed on the first line of the next page of the source listing.

20.2.3.18 %PROCEDURE Statement

The %PROCEDURE statement begins a series of preprocessor statements that constitute a preprocessor procedure. Preprocessor procedures are function procedures that may occur anywhere in a source program.

The format of the %PROCEDURE statement is

```
%label:PROCEDURE [(parameter-identifier,...)]
[STATEMENT]
RETURNS(
                ⎧ CHARACTER ⎫
                ⎨ FIXED     ⎬ );
                ⎩ BIT       ⎭
        .
        .
        .
[%]RETURN (preprocessor-expression);
        .
        .
        .
[%]END;
```

label

An unsubscripted label constant. A preprocessor procedure is invoked by the appearance of the label name on the %PROCEDURE statement and terminated by the corresponding %END statement.

The label name must be active if invoked from a nonpreprocessor statement.

Preprocessor label names may be activated and deactivated, but may not be specified in a %DECLARE statement.

parameter-identifier

The name of a preprocessor identifier. Each identifier is a parameter of the procedure.

RETURNS

A preprocessor procedure attribute. The RETURNS attribute defines the data type to be returned to the point of invocation in the source code. If you specify a data type that is inconsistent with the returned value, a conversion error may result.

STATEMENT

A preprocessor procedure option. The STATEMENT option permits the use of a keyword argument list followed by an optional positional argument list in the preprocessor procedure invocation. For further information, see "Using the STATEMENT option" below.

preprocessor-expression

Value to be returned to the invoking source code. The preprocessor expression must be specified. The preprocessor expression is converted to the data type specified in the RETURNS option and is returned to the point of invocation. Therefore, the expression must be capable of being converted to CHARACTER(255), FIXED(10), or BIT(31).

Preprocessor procedures may not be nested. The scope of a preprocessor procedure is the procedure itself; that is, variables, labels, and any %GOTO statements used inside of the procedure must be local.

When a preprocessor procedure (with or without the STATEMENT option) is invoked from a preprocessor statement, each argument is treated as an expression and the result of executing the preprocessor procedure is returned to the statement containing the invocation.

When a preprocessor procedure is invoked from nonpreprocessor source text, the arguments are interpreted as character strings and are delimited by the appearance of a comma or a right parenthesis occurring outside of balanced parentheses. For example, the positional argument list (Q(E,D), XYZ) has two arguments; the strings 'Q(E,D)' and 'XYZ'.

The following example declares the preprocessor procedure A1 and specifies that the procedure return a fixed decimal result after the preprocessor statements within the procedure have been executed:

```
%A1: PROCEDURE RETURNS(FIXED);
     DECLARE (A,B,C) FIXED;

        A = 2;
        B = 10;
        X = A + B;
        RETURN(C);
     END;
```

This example returns the value 12 to the point of invocation. Note that the leading percent signs, normally associated with preprocessor statements, are not required within a preprocessor procedure.

The next example uses a preprocessor procedure to return a Fibonacci number.

```
PPFIB: PROCEDURE OPTIONS (MAIN);
       DECLARE Y CHAR(14) INITIAL('Fibonacci Test');  ❶
       %DECLARE Y FIXED;  ❷
       %F: PROCEDURE(X) RETURNS (FIXED);  ❸
          DECLARE X FIXED;
          IF (X <= 1)
             THEN RETURN(1);
             ELSE RETURN(F(X-1)+F(X-2));
          END;          /* End preprocessor procedure */
       %Y = F(10);  ❹
       PUT SKIP LIST(Y);
       %Y = F(11);  ❺
       PUT SKIP LIST(Y);
       %Y = F(12);  ❻
       PUT SKIP LIST(Y);
       %DEACTIVATE Y;  ❼
       PUT SKIP LIST(Y);  ❽
       END;             /* End run-time procedure */
```

In this example, the recursive preprocessor procedure labeled %F is invoked to return a single value, a Fibonacci number, to the point of invocation. The following notes correspond to the example:

❶ The run-time variable ′Y′is declared with the CHARACTER attribute and initialized to ′Fibonacci Test′.

❷ The preprocessor variable ′Y′ is declared with the FIXED attribute, which implies FIXED DECIMAL (10,0). This declaration automatically activates the preprocessor variable ′Y′.

❸ The preprocessor procedure ′F′ is defined. The percent sign for the END statement is optional in a preprocessor procedure.

Note that this procedure is recursive, that is, it invokes itself.

❹ The preprocessor procedure is called, passed the value 10, and the 10th number in the Fibonacci series is calculated. The resulting value is assigned to the preprocessor variable ′Y.′

Since the preprocessor variable ′Y′ is active by default, the compiler replaces the occurrence of ′Y′ in the PUT statement with the new preprocessor ′Y′ value.

❺ Step 4 is repeated for the value 11.

❻ Step 4 is repeated for the value 12.

❼ The preprocessor variable ′Y′ is deactivated. No more scanning or replacement occurs. The preprocessor variable ′Y′ retains its final replacement value, 233.

❽ The run-time value of ′Y′ (′Fibonacci Test′) is output.

The output from this program is

```
89
144
233
Fibonacci Test
```

For an example of a preprocessor procedure that uses the STATEMENT option, see Section 20.2.2.

20.2.3.19 %REPLACE Statement

The preprocessor %REPLACE statement specifies that an identifier is a constant of a given value. It may be used anywhere within a procedure or anywhere in a PL/I source file. However, note that you may not use the same identifier as both a %REPLACE identifier and a declared preprocessor identifier or program variable.

Beginning at the point at which a %REPLACE statement is encountered, PL/I replaces all occurrences of the specified identifier with the specified constant value, until the end of compilation.

The format of the %REPLACE statement is

```
%REPLACE identifier BY constant-value;
```

identifier

Any valid PL/I identifier. PL/I keywords are not valid identifiers in a %REPLACE statement. The identifier must not be the name of a declared preprocessor or program variable. VAX–11 PL/I permits multiple %REPLACE statements and %REPLACE statements that redefine the %REPLACE identifier.

constant-value

Any valid character-string, bit-string, or arithmetic constant.

Integer constants that are given values by %REPLACE statements are valid in constant expressions. For example:

```
%REPLACE PREFIX BY 8;

DECLARE BUFFER CHARACTER (80 + PREFIX);
```

When the program containing these lines is compiled, the variable BUFFER is declared with a length of 88 characters.

20.2.3.20 %RETURN Statement

The %RETURN statement terminates execution of the current preprocessor procedure. The format of the %RETURN statement is

```
[%]RETURN (preprocessor-expression);
```

preprocessor-expression

Value to be returned to the invoking procedure. The preprocessor expression must be specified. The preprocessor expression is converted to the data type specified in the RETURNS option, and the value of the expression is returned to the point of invocation. Therefore, it must be capable of being converted to CHARACTER, FIXED, or BIT.

The value returned by a preprocessor procedure may not itself contain preprocessor statements.

The value of the evaluated preprocessor expression is passed back to the point of invocation, and control returns to the evaluation of the source code statement that contained the reference to the preprocessor procedure. The maximum precision of the value returned by %RETURN is BIT(31), CHARACTER(255), and FIXED(10).

Within a preprocessor procedure, the leading percent (%) sign is optional because preprocessor procedures do not require percent signs within the procedure.

Multiple %RETURN statements are permitted in preprocessor procedures.

20.2.3.21 %SBTTL Statement

The %SBTTL statement allows specification of an arbitrary compile-time string for the listing subtitle line. PL/I uses the procedure IDENT, or V002 if no IDENT was specified. If %SBTTL is used, the specified subtitle appears to the right of IDENT or V002.

The format of the %SBTLL statement is

```
%SBTTL preprocessor-expression
```

preprocessor-expression

A maximum of 30 characters giving the listing subtitle. Subtitles of more than 30 characters are truncated.

20.2.3.22 %TITLE Statement

The %TITLE statement allows specification of an arbitrary compile-time string for the listing title line. If %TITLE is used, the specified title appears to the right of the customary title. (If no TITLE option is specified, PL/I uses the name of the first level-1 procedure in the source program as the title.)

The format of the %TITLE statement is

```
%TITLE preprocessor-expression
```

preprocessor-expression

A maximum of 30 characters giving the listing title. Titles of more than 30 characters are truncated.

20.2.3.23 %WARN Statement

The %WARN statement provides a diagnostic warning message during program compilation. The format of the %WARN statement is

```
%WARN preprocessor-expression;
```

preprocessor-expression

A maximum of 64 characters giving the text of the warning message to be displayed. Messages of more than 64 characters are truncated.

The returned message displayed by %WARN is

```
%PLIG-W-USERDIAG, text
```

text

The preprocessor expression specified by the %WARN statement in the source program.

The %WARN statement increments the warning diagnostic count displayed in the compilation summary.

20.2.4 Preprocessor Built-In Functions

A number of PL/I built-in functions are available for use at compile time. With few exceptions, they have the same effect as run-time PL/I built-in functions with the same name.

The preprocessor built-in functions are summarized in Table 20-2, according to the following functional categories:

- Arithmetic built-in functions—functions that provide information about the properties of arithmetic values, or that perform common arithmetic calculations
- String-handling built-in functions—functions that process character- and bit-string values
- Conversion built-in functions—functions that convert data from one data type to another
- Timekeeping built-in functions—functions that return the system date and time of day
- Miscellaneous—functions that are specifically preprocessor built-in functions

Table 20-2: Summary of PL/I Preprocessor Built-In Functions

Category	Function Reference	Value Returned
Arithmetic	ABS(x)	Absolute value of x
	MAX(x1,x2)	Larger of the values x1 and x2
	MIN(x1,x2)	Smaller of the values x1 and x2
	MOD(x,y)	Value of x modulo y
	SIGN(x)	-1, 0, or 1 to indicate the sign of x
String-Handling	COPY(s,c)	c copies of specified string s
	INDEX(s,c)	Position of the character string c within the string s
	LENGTH(s)	Number of characters or bits in the string s
	SEARCH(s,c)	Position of the first character in s that is found in c
	SUBSTR(s,i,[,j])	Part of string s beginning at i for j characters
	TRANSLATE(s,c[,d])	String s with substitutions defined in c and d
	TRIM(s[,e,f])	String s with all characters in e removed from the left and all characters in f removed from the right
	VERIFY(s,c)	Position of the first character is s which is not found in c
Conversion	BYTE(x)	ASCII character represented by the integer x
	RANK(c)	Integer representation of the ASCII character c
Timekeeping	DATE( )	System date of compilation in the form YYMMDD
	TIME( )	System time of day of compilation in the form HHMMSSXX
Miscellaneous	ERROR( )	Count of user-generated diagnostic error messages
	INFORM( )	Count of user-generated diagnostic informational messages
	LINE( )	Line number in source program that contains the end of a specified preprocessor statement
	VARIANT( )	String result representing the value of the /VARIANT PLI command qualifier
	WARN( )	Count of user-generated diagnostic warning messages

Except for those in the category labeled 'Miscellaneous, preprocessor built-in functions are similar to their nonpreprocessor counterparts. Miscellaneous functions are summarized in the next sections. See Chapter 19 for explanations of all other built-in functions.

20.2.4.1 ERROR Preprocessor Built-In Function

The preprocessor ERROR built-in function returns the number of preprocessor diagnostic error messages issued during compilation up to that particular point in the source program. The format for the ERROR built-in function is

```
ERROR();
```

The function returns a FIXED result representing the number of compile-time warning messages that were issued up to the point at which the %ERROR statement was encountered.

20.2.4.2 INFORM Preprocessor Built-In Function

The preprocessor INFORM built-in function returns the number of diagnostic informational messages issued during compilation up to that particular point in the source program. The format for the INFORM built-in function is

```
INFORM()
```

The function returns a FIXED result representing the number of compile-time warning messages that were issued up to the point at which the %INFORM statement was encountered. See Section 20.3.1 for an example.

20.2.4.3 LINE Preprocessor Built-In Function

The LINE preprocessor built-in function returns the line number of the source program text containing the end of the preprocessor statement that calls the LINE built-in function. The line number is returned as a FIXED integer.

The format of the function within a preprocessor expression is

```
LINE()
```

20.2.4.4 VARIANT Preprocessor Built-In Function

The VARIANT preprocessor built-in function returns a string representing the value of the /VARIANT qualifier in the PLI command that invoked the compilation.

Its format in a preprocessor expression is

```
VARIANT()
```

The /VARIANT qualifier permits specification of compilation variants.

The format of compilation variants is

```
/VARIANT [ =alphanumeric-string   ]
         [ ="alphanumeric-string" ]
```

For example, if a program may be compiled with a choice of three different INCLUDE files, you can use the /VARIANT command qualifier to specify which file is to be included. In the following example, the file 'SPECIAL.SRC' is included in the program only if /VARIANT=SPECIAL appears in the PLI command line:

```
%IF VARIANT() = 'SPECIAL'
%THEN
    %INCLUDE 'SPECIAL.SRC';
%IF VARIANT() = 'NONE'
%THEN;
```

No action is taken if /VARIANT=NONE appears on the PLI command line.

If /VARIANT is not specified, or if it is specified without a value, the default value is /VARIANT ="".

20.2.4.5 WARN Preprocessor Built-In Function

The preprocessor WARN built-in function returns the number of diagnostic warning messages issued during compilation up to that particular point in the source program. The format for the WARN built-in function is

```
WARN()
```

The function returns a fixed result representing the number of compile-time warning messages that were issued up to the point at which the %WARN statement was encountered.

Appendix A
Rules for Conversion of Data

This appendix provides details of the data type conversions that PL/I performs when assigning values to variables. The rules for conversions apply to

- Assignment statements.
- Arguments passed to a procedure.
- Values specified in a RETURN statement.
- Arguments converted by the built-in functions FIXED, FLOAT, BINARY, DECIMAL, BIT, or CHARACTER.
- Character-string arguments to the PUT and GET statements.

A.1 Assignments to Arithmetic Variables

You can assign expressions of any computational type to arithmetic variables. Note, however, that the compiler may issue a warning message unless an explicit conversion function is used. The conversion rules are described below for each source type.

A.1.1 Arithmetic to Arithmetic Conversions

You can assign a source expression of any arithmetic type to a target variable of any arithmetic type. Note the following qualifications:

- If the target is a variable of type FIXED BINARY or FIXED DECIMAL, then the FIXEDOVERFLOW condition is signaled when the source value has a larger number of integral digits than are specified in the precision of the target. If the target is a fixed-point binary variable, FIXEDOVERFLOW is signaled if the source value exceeds the storage allocated for the target.
- If the target is a variable of type FIXED-POINT(p,q) and the source value has more than q fractional digits, then the excess fractional

digits of the source are truncated, and no condition is signaled. If the source has fewer than q fractional digits, the source value is padded on the right with zeros.

- If the target value is floating point and the absolute source value is too large to be represented by a VAX floating-point type (see Section 8.2.3), then the OVERFLOW condition is signaled, and the value of the target is undefined. If the absolute source value is too small to be represented, the value zero is assigned to the target, and, if enabled, the UNDERFLOW condition is signaled.

A.1.2 Pictured to Arithmetic Conversions

In VAX-11 PL/I, all pictured values have the associated attributes FIXED DECIMAL(p,q), where p is the total number of characters in the picture specification that specify decimal digits, and q is the total number of these digits that occur to the right of the V character. If the picture specification does not include a V character, then q is zero. This associated fixed-point decimal value is assigned to the target, following the rules for arithmetic to arithmetic conversion described above.

A.1.3 Bit-String to Arithmetic Conversions

When a bit-string value is assigned to an arithmetic variable, PL/I treats the bit string as a nonnegative fixed-point binary value. If the converted value is greater than or equal to 2^{31}, then FIXEDOVERFLOW is signaled. The leftmost bit in the bit string (as output by PUT LIST) is the most significant bit in the fixed-point binary value, not its sign. If the bit string is null, the fixed-point binary value is zero. The intermediate fixed-point binary value is then converted to the target arithmetic type.

Note that a bit string interpreted as a fixed-point binary value changes its value when assigned to a bit-string variable of a different length. See Section 8.4.4 for further details.

A.1.4 Character String to Arithmetic Conversions

When a character string is assigned to an arithmetic value, PL/I interprets the string as an arithmetic constant and creates an intermediate numeric value based on the characters in the string. The string can contain any series of characters that describes a valid arithmetic constant. If it contains any invalid characters, the ERROR condition is signaled.

PL/I then converts the intermediate value to the data type of the target, following the rules for arithmetic to arithmetic conversions. In conversions to fixed point, FIXEDOVERFLOW is signaled if the character string speci-

fies too many integral digits. Excess fractional digits are truncated without signaling a condition.

If the source character string is null or contains all spaces, the resulting arithmetic value is zero.

A.2 Assignments to Bit-String Variables

In the conversion of any data type to a bit string, PL/I first converts the source data item to an intermediate bit-string value. Then, based on the length of the target string, it performs one of the following:

- If the length of the target bit-string value is greater than the length of the intermediate string, the target bit string (as represented by PUT LIST) is padded with zeros on the right.
- If the length of the target bit-string value is less than the length of the intermediate string, the intermediate bit string (as represented by PUT LIST) is truncated on the right.

The next sections describe how PL/I arrives at the intermediate bit-string value for each data type.

A.2.1 Arithmetic and Pictured to Bit-String Conversions

In converting an arithmetic value to a bit-string value, PL/I first computes the absolute value of the arithmetic value, and then converts it to an integer of type FIXED BINARY with a maximum precision of 31. If this conversion results in an integer larger than the data type can accommodate, the FIXEDOVERFLOW condition is signaled; otherwise, each of the bits of the intermediate bit string represents a binary digit of n.

During the conversion, the sign of the arithmetic value and any fractional digits are lost. As a result, a value that contains only fractional digits (such as 0.2312) is converted to an all-zero bit string.

If an arithmetic value is assigned to a bit-string variable, and that variable is assigned to a second variable of different length, the effect is to multiply or divide the arithmetic value as a result of padding or truncating the bit string. See Section 8.4.4 for further details.

A.2.2 Character-String to Bit-String Conversions

PL/I can convert a character string of 0s and 1s to a bit string. Any character in the character string other than 0 or 1, including spaces, will signal the ERROR condition. If the source is a null character string, the intermediate string is a null bit string.

A.3 Assignments to Character-String Variables

In the conversion of any data type to a character string, PL/I first converts the source value to an intermediate character-string value. Then it performs one of the following:

- If the length of the intermediate string is zero, a null string is assigned to the target.
- If the target is a returns descriptor with an asterisk extent (as in RETURNS CHAR(*)), the intermediate string is assigned to the target.
- If the intermediate string is shorter than the maximum length of the target, and the target has the VARYING attribute, it is assigned the value of the intermediate string without trailing spaces. If the target does not have the VARYING attribute, the string is padded with trailing spaces.
- If the maximum length of the target character string is less than the length of the intermediate string, the intermediate string is truncated.

The next sections describe how PL/I arrives at the intermediate string for conversion of each data type. Examples show the intermediate value, as well as the resulting value.

A.3.1 Arithmetic to Character-String Conversions

The manner in which PL/I converts the arithmetic item depends on the data type of the source, as described below.

A.3.1.1 Conversion from Fixed-Point Binary or Decimal

If the source value is of type FIXED BINARY, PL/I first converts it to type FIXED DECIMAL. PL/I converts a value with attributes FIXED DECIMAL to an intermediate string with the numeric value right justified in the string. The format of the intermediate string can be described as follows:

- If there are no fractional digits, the first two characters of the string are spaces. The last characters in the string are the digit characters representing all the digits in the integer; leading zeros are replaced by spaces except in the last position. If the integer is negative, a minus sign immediately precedes the first digit; if not, this position contains a space. At least one digit always appears, in the last position in the string.
- If there are no integral digits, the first three characters are (in order) an optional minus sign if the fraction is negative, the digit 0, and a

decimal point. If the number is not negative, the first character is a space. The last characters in the string are all the fractional digits of the number.

- If there are both integral and fractional digits, the first character is always a space. The last characters are all the fractional digits of the number and are preceded by a decimal point; the decimal point is always preceded by at least one digit, which may be 0; all integral digits appear before the decimal point, and leading zeros are replaced by spaces. A minus sign precedes the first integral digit if the number is negative; if not, then the minus sign is replaced by a space.

These rules may cause confusion if you do not take into account the leading spaces. In the following examples, the letter b represents a space:

```
DECLARE STRING1 CHARACTER (8),
        STRING2 CHARACTER (4);

STRING1 = 283472.;
        /* intermediate string = 'bbb283472',
        STRING1 = 'bbb28347' */

STRING2 = 283472.;
        /* intermediate string = 'bbb283472',
        STRING2 = 'bbb2' */

STRING2 = -283472.;
        /* intermediate string = 'bb-283472',
        STRING2 = 'bb-2' */

STRING2 = -.003344;
        /* intermediate string = '-0.003344',
        STRING2 = '-0.0' */

STRING2 = -283.472;
        /* intermediate string = 'b-283.472',
        STRING2 = 'b-28' */

STRING2 = 283.472;
        /* intermediate string = 'bb283.472',
        STRING2 = 'bb28' */
```

A.3.1.2 Conversion from Floating-Point Binary or Decimal

If the source value is of type FLOAT BINARY, it is converted to FLOAT DECIMAL. For a value of type FLOAT DECIMAL(p), where p is less than or equal to 34, the intermediate string is of length p+6; this allows extra characters for the sign of the number, the decimal point, the letter E, the sign of the exponent, and the 2-digit exponent.

NOTE

> If the value is a floating-point number of the VAX type G-float, three characters are allocated to the exponent, and the length of the string is p+7. If the value is of type H-float, four characters are allocated to the exponent, and the length of the string is p+8.

If the number is negative, the first character is a minus sign; otherwise, the first character is a space. The subsequent characters are a single digit (which may be 0), a decimal point, p–1 fractional digits, the letter E, the sign of the exponent (always + or –), and the exponent digits. The exponent field is of fixed length, and the zero exponent is shown as all zeros in the exponent field.

For example:

```
CONCH: PROCEDURE OPTIONS(MAIN);

DECLARE OUT PRINT FILE;

OPEN FILE(OUT) TITLE('CONCH.OUT');

PUT SKIP FILE(OUT) EDIT('''',25E25,'''') (A);
PUT SKIP FILE(OUT) EDIT('''',-25E25,'''') (A);
PUT SKIP FILE(OUT) EDIT('''',1.233325E-5,'''') (A);
PUT SKIP FILE(OUT) EDIT('''',-1.233325E-5,'''') (A);

END CONCH;
```

The program CONCH produces the output

```
' 2.5E+26'
'-2.5E+26'
' 1.233325E-05'
'-1.233325E-05'
```

The PUT statement converts its output sources to character strings, following the rules described in this section. (The output strings have been surrounded with apostrophes to make the spaces distinguishable.) Note that, in each case, the width of the quoted output field (that is, the length of the converted character string) is the precision of the floating-point constant plus 6.

A.3.2 Pictured to Character-String Conversions

If the source value is pictured, its internal, character-string representation is used without conversion as the intermediate character string.

A.3.3 Bit-String to Character-String Conversions

When PL/I converts a bit string to a character string, it converts each bit (as represented by PUT LIST) to a 0 or 1 character in the corresponding position of the intermediate character string.

If the bit string is a null string, the intermediate character string is also null.

A.4 Assignments to Pictured Variables

A source expression of any computational type can be assigned to a pictured variable. The target pictured variable has a precision (p), which is defined as the number of characters in its picture specification that specify decimal digits. It also has a scale factor (q), which is defined as the number of picture characters that specify digits and occur to the right of the V character in the picture specification. If there is no V character, or if all digit-specification characters are to the left of V, then q is zero.

The source expression is converted to a fixed-point decimal value v of precision (p,q), following the rules given in Section A.1 for conversion from the source data type to fixed decimal. This value is then edited to a character string s, as specified by the picture specification, and the value s is assigned to the pictured target.

When the value v is being edited to the string s, the ERROR condition is signaled if the value of v is less than zero and the picture specification does not contain one of the characters S, +, -, T, I, R, CR, or DB. The value of s is then undefined. FIXEDOVERFLOW is also signaled if the value v has more integral digits than are specified by the picture specification of the target.

A.5 Conversions Between Offsets and Pointers

Offset variables are given values by assignment from existing offset values or from conversion of pointer values. Pointer variables are given values by assignment from existing pointer values or from conversion of offset values.

The OFFSET built-in function converts a pointer value to an offset value. The POINTER built-in function converts an offset value to a pointer. These functions are described in Section 19.2.

PL/I also automatically converts a pointer value to an offset value, and vice versa, in an assignment statement. The following assignments are valid:

1. pointer-variable = pointer-value;
2. offset-variable = offset-value;
3. pointer-variable = offset-variable;
4. offset-variable = pointer-value;

In (3) and (4), above, the offset variable must have been declared with an area reference.

Appendix B
Calling System Services

System services are procedures implemented by the VAX/VMS operating system. They provide timer, I/O, and other system-related functions that are otherwise unavailable to the VAX-11 PL/I programmer. Although the use of some system services is restricted by privilege requirements, many are available for general programming use. System services are described in detail in the *VAX/VMS System Services Reference Manual.*

This appendix provides information on

- Declaring system services in PL/I.
- Specifying arguments for system services.
- Testing status values returned from system services.

Section B.4 contains examples of calling system services from PL/I programs.

B.1 Declaring System Services

The default PL/I text library PLISYSDEF.TLB contains declarations for all system services as external entries that return FIXED BINARY (31) values. The text module names have the form

```
SYS$name
```

where SYS$name is the name of the system service. Thus, to declare a system service you are going to use, specify an %INCLUDE statement as in this example:

```
%INCLUDE SYS$TRNLOG;
```

The compiler, by default, locates the module SYS$TRNLOG in PLISYSDEF.TLB during compilation. This module contains a complete declaration for an external entry constant that invokes the Translate Logical Name system service, including the appropriate parameter descriptor. The examples in Section B.4 show the texts of several such declarations.

B.2 Specifying Arguments for System Services

When you specify arguments for a system service, you must determine from its description the following information about each argument:

- Which mechanism must be used to pass the argument
- What the data type of the argument is
- Whether the argument can be omitted
- Whether you need to use system global symbols to specify an argument

To examine the PL/I parameter descriptors in the declaration of a given system service, you can display or print the text module for that service. For example:

```
$ LIBRARY/EXTRACT=SYS$TRNLOG/OUTPUT=LP:TRNLOG -
$_SYS$LIBRARY:PLISYSDEF/TEXT
```

This LIBRARY command prints the contents of the text module SYS$TRNLOG from the library SYS$LIBRARY:PLISYSDEF. The file is printed on the device LP; the listing file is named TRNLOG.TXT.

B.2.1 Argument-Passing Mechanisms Used by System Services

There are three ways that a PL/I procedure can pass an argument to a non-PL/I procedure such as a system service. They are

- By immediate value. When an argument is passed by immediate value, the actual value of the argument is passed.
- By reference. When an argument is passed by reference, the address in storage of the argument is passed.
- By descriptor. When an argument is passed by descriptor, the address in storage of a data structure describing the argument is passed.

These methods are described in Section 13.2.2. System services use the following methods of passing arguments:

- Input arguments that are either fixed binary or bit-string values and that can be expressed in 32 bits are always passed by immediate value.
- Input arguments that cannot be expressed in 32 bits are passed by reference.

- Output arguments are always passed by reference.
- Character-string arguments, either input or output, are always passed by character-string descriptor.

B.2.2 Parameter Descriptors for System Services Data Types

Tables B-1 and B-2 list the types of input and output parameters used by system services. Each table shows the parameter descriptor for a type of argument.

Table B-1: Input Arguments for System Services

Argument Data Type	Parameter Declaration in PLISYSDEF
Numeric values:	FIXED BINARY(31) VALUE
Indicator	
Channel number	
Event flag	
Access mode	
Binary mask	
Buffer size	
Process identification	
UIC	
Character strings:	CHARACTER(*)
Logical name	
Process name	
Device name	
Cluster name	
Time string	
Bit masks:	
32 bits or less	BIT(32) ALIGNED VALUE
64 bits	BIT(64) ALIGNED
Time values	BIT(64) ALIGNED
Entry mask or routine	ENTRY VALUE
Buffers:	ANY
Item list	
Quota list	
AST parameter	ANY VALUE
Two-longword array	(2) FIXED BINARY(31)

Table B-2: Output Arguments for System Services

Argument Data Type	Parameter Declaration in PLISYSDEF
Numeric values:	
String length	FIXED BINARY(15)
Channel number	FIXED BINARY(15)
Access mode	FIXED BINARY(7)
Table number	FIXED BINARY(7)
Process identification	FIXED BINARY(31)
Character strings:	CHARACTER(*)
Equivalence name	
Time string	
Buffers:	ANY
I/O status block	
Time value	BIT(64) ALIGNED
Two-longword array	(2) FIXED BINARY(31)

B.2.3 Variable-Length Argument Lists

Many system services provide default values when a parameter contains a zero. In VAX-11 PL/I, you can omit an optional argument for a system service, and the compiler will pass a zero for the argument. The system services are all declared with OPTIONS (VARIABLE) to allow this to occur.

An argument list must, however, always contain the number of commas that would be present if all arguments were specified. For example:

```
STS$VALUE = SYS$GETJPI(,,,JPI_LIST,,,,);
```

In this example, the SYS$GETJPI system service has eight parameters, but only one need be specified. Commas indicate the omitted arguments.

If an argument list does not specify the required number of arguments (or commas), a function reference to the system service returns the value associated with the status code SS$_INSFARGS (meaning insufficient arguments).

B.2.4 Symbol Definitions for System Service Arguments

Many system services require you to specify values using binary integers or bit-string masks for which symbolic names exist. Symbolic names are more meaningful within a program listing and, should software be updated, will not change in meaning even if their corresponding values change.

In PL/I, you can declare the names of global symbols using the GLOBALREF and VALUE attributes. Then you may use the names to represent values in an argument list for a system service invocation. For example:

```
DECLARE (IO$_READVBLK, IO$M_TIMED, IO$M_CVTLOW)
        FIXED BINARY (31) GLOBALREF VALUE;

%INCLUDE SYS$QIOW;
%INCLUDE $STSDEF;
            .
            .
            .
        STS$VALUE = SYS$QIOW (1,TTCHAN,
            IO$_READVBLK+IO$M_TIMED+IO$M_CVTLOW,
            IOSB,,,ADDR(COMMAND),3,DELTA,,,);
```

In this example, IO$_READVBLK, IO$M_TIMED, and IO$M_CVTLOW are declared as global symbols. They are bit masks that are meaningful to the Queue I/O Request system service. When combined in the invocation of SYS$QIOW, they request the system to read a virtual block, place a specified time limit on the operation, and convert the received characters to uppercase.

B.3 Testing Return Values from System Services

All system services return a longword integer value with the severity of the return status indicated in the low-order three bits and the success/failure indicated in the low-order bit.

When you write references to system services, it is your responsibility to provide tests for successful completion of the service. Neither the services themselves nor the run-time error reporting procedures provide run-time messages for system service errors. You can test for successful completion in the following ways:

- Test for success/failure by testing the low-order bit of the return status value.
- Test for specific return values by declaring the global symbol names associated with each return status value.

The text module $STSDEF contains the declarations for the variables STS$VALUE and STS$SUCCESS. STS$VALUE is an integer variable; STS$SUCCESS is a 1-bit variable that is based on the low-order bit of STS$VALUE. Thus, these variables can be used to test both the specific return value and the low-order bit. To gain access to STS$VALUE and STS$SUCCESS, place the following statement in your program:

```
%INCLUDE $STSDEF;
```

System global symbol names are defined for all status values returned by system services. These names have the format

SS$_condition

where condition is a mnemonic that describes the condition.

The definitions for all of these global symbols are located in the default system library in SYS$LIBRARY. In a PL/I program, to test function returns from system services you must declare the names as follows:

```
DECLARE SS$_condition GLOBALREF
      FIXED BINARY(31) VALUE;
```

For example:

```
%INCLUDE SYS$SETEF;
DECLARE SS$_WASSET FIXED BINARY (31) GLOBALREF VALUE;
%INCLUDE $STSDEF;

STS$VALUE = SYS$SETEF(4);

IF ^STS$SUCCESS THEN RETURN (STS$VALUE);
IF STS$VALUE = SS$_WASSET THEN DO;
```

In this example, the return status value from the call to SYS$SETEF is placed in STS$VALUE. If STS$SUCCESS, the low-order bit of STS$VALUE, is false, an error has occurred; the procedure then returns the value of STS$VALUE to its caller. If STS$SUCCESS is true, the procedure tests whether STS$VALUE is equal to the value of SS$_WASSET and continues.

B.4 Examples of System Services

The examples on the next few pages contain a number of system service calls. These examples illustrate

- Translating a logical name.
- Using timer and time conversion routines.

All the sample programs use the system service INCLUDE files in PLISYSDEF to declare the system services. The text of each sample program shows the INCLUDE file for the system service.

All procedures also include the module $STSDEF; however, the contents of this text module are not shown in the examples.

B.4.1 Logical Name Translation

This program illustrates a call to the Translate Logical Name (SYS$TRNLOG) system service, which returns the result of a single logical

name translation. In this example, the procedure ORION translates the logical name CYGNUS and displays the result on the terminal. If the name is not defined, the procedure displays a message indicating that fact. The circled numbers are keyed to the notes below.

```
ORION: PROCEDURE RETURNS(FIXED BINARY(31));

%INCLUDE SYS$TRNLOG;
/* Translate Logical Name system service */
declare sys$trnlog external entry (                              ❶
        char(*);                  /* logical name string */
        fixed bin(15),            /* variable to receive translated length */
        char(*),                  /* variable to receive translated name */
        fixed bin(7),             /* variable to receive table number */
        fixed bin(7),             /* variable to receive access mode */
        fixed bin(31) value)      /* variable search disable mask */

        options(variable) returns(fixed bin(31));
%INCLUDE $STSDEF;

DECLARE CYGDES CHARACTER(6) STATIC INITIAL('CYGNUS'),

        NAMEDES CHARACTER(63),
        NAMELEN FIXED BINARY(15);     ❷

DECLARE SS$_NOTRAN GLOBALREF FIXED BINARY(31) VALUE;  ❸

        STS$VALUE = SYS$TRNLOG(CYGDES,NAMELEN,NAMEDES,,,,);   ❹
        IF STS$VALUE = SS$_NOTRAN THEN
              PUT SKIP LIST('CYGNUS not defined');    ❺
        ELSE                          ❻
           IF STS$SUCCESS THEN
              PUT LIST('CYGNUS is',
                      SUBSTR(NAMEDES,1,NAMELEN));
        RETURN(STS$VALUE);   ❼
        END;
```

❶ SYS$TRNLOG's first three parameters are input and output character-string descriptors and a field in which the service returns the length of the character string returned. The remaining parameters are optional.

❷ The procedure declares the logical name to be translated and an output character-string buffer, NAMEDES, for the translated name. The variable declared for the output character string must not be VARYING. The variable NAMELEN is FIXED BINARY(15) to match the parameter descriptor.

❸ The procedure declares the global symbol value SS$_NOTRAN. This value is returned from SYS$TRNLOG if no logical name assignment exists.

❹ The reference to SYS$TRNLOG specifies the logical name CYGNUS and the variables to receive the translated logical name length and logical name. The procedure reference does not specify the final arguments. At run time, the argument list for this procedure will contain zeros for these arguments.

❺ On return from SYS$TRNLOG, the variable STS$VALUE is compared with the value of the global symbol SS$_NOTRAN. If they match, the procedure displays a message indicating there is no logical name assignment.

❻ If they do not match, the procedure checks whether the service completed successfully. If so, it displays the equivalence name returned.

❼ In either case, the procedure exits with the status value returned by SYS$TRNLOG. If this is an error value, the command interpreter will display a message.

B.4.2 Timer and Time Conversion Routines

The system services for performing activity based on time, either an absolute time or a delta time, refer to a time value that is maintained in a 64-bit field. There are system services that convert a character string that specifies a time to its binary equivalent and vice versa.

B.4.2.1 Obtaining a Time Value in System Format

The PL/I procedure GETBINTIM, shown in the next program, accepts a character-string time value as a parameter and returns the binary time value to its point of invocation. The circled numbers are keyed to the notes below.

```
/*
  This procedure converts a time given in ASCII format to a
  64-bit time value that is used internally by VAX/VMS.
  Input strings must be of the form:

  dd-mmm-yyyy hh:mm:ss.cc (for an absolute date or time)
  dddd hh:mm:ss.cc        (for a delta time)
*/
GETBINTIM: PROCEDURE (ASCII_STRING) RETURNS(BIT(64) ALIGNED);

%INCLUDE SYS$BINTIM;
/* Convert ASCII String to Binary Time system service */
declare sys$bintim external entry (                          ❶
        char(*),               /* ASCII string */
        bit(64) aligned)       /* variable to receive system time */

        returns (fixed binary(31));
%INCLUDE $STSDEF;

DECLARE ASCII_STRING CHARACTER(*),
        BINARY_TIME BIT(64) ALIGNED;

        STS$VALUE = SYS$BINTIM(ASCII_STRING,BINARY_TIME);   ❷
/*
  If successful, return binary time to point of invocation. Otherwise,
  return 0 - this results in absolute time 17-NOV-1858.
*/
       IF STS$SUCCESS THEN RETURN(BINARY_TIME);
       ELSE RETURN(0);
END;
```

❶ GETBINTIM declares the system service SYS$BINTIM, which converts an ASCII string to a binary time value. SYS$BINTIM requires an input character-string descriptor, declared as CHARACTER(*), and the address of a variable to receive the converted 64-bit time value.

❷ GETBINTIM invokes SYS$BINTIM as a function and tests the return status. An error results if the ASCII time value is not specified correctly. When an error is returned, GETBINTIM returns a zero to its point of invocation.

This procedure may be invoked as follows, to supply a date and time value for a file in an ENVIRONMENT option:

```
DECLARE GETBINTIM ENTRY( CHAR(*) ) RETURNS BIT(64) ALIGNED,
        (CREATED_DATE,EXPIRE_DATE) BIT(64) ALIGNED;

     CREATED_DATE = GETBINTIM('15-MAY-1981 00:00:00.00');
     EXPIRE_DATE = GETBINTIM('31-DEC-1981 00:00:00.00');
     OPEN FILE(TAPEFILE) ENVIRONMENT(

             CREATION_DATE(CREATED_DATE),
             EXPIRATION_DATE(EXPIRE_DATE));
```

B.4.2.2 Setting the Timer

The procedure in the next program uses the Set Timer (SYS$SETIMR) system service. It issues a time request for some activity to occur in 10 seconds and specifies the number of an event flag to be set when the 10 seconds have elapsed. The circled numbers are keyed to the notes below.

```
SET_TIMER: PROCEDURE OPTIONS(MAIN) RETURNS
                      (FIXED BINARY(31));

%INCLUDE SYS$CLREF;
/* Clear Event Flag system service */
declare sys$clref external entry (
        fixed bin(31) value)   /* event flag number */

        returns (fixed bin(31));

%INCLUDE SYS$SETIMR;
/* Set Timer system service */
declare sys$setimr external entry (
        fixed bin(31) value,   /* event flag number */
        bit(64) aligned,       /* time value */
        entry value,           /* external AST procedure */
        fixed bin(31) value)   /* AST parameter */

        options(variable) returns(fixed bin(31));
%INCLUDE SYS$WAITFR;
/* Wait for Event Flag system service */
declare sys$waitfr external entry (
        fixed bin(31) value)   /* event flag */

        returns (fixed bin(31));
%INCLUDE $STSDEF;
DECLARE GETBINTIM ENTRY (
                        CHAR(*))         /* character-string time */
                RETURNS (BIT(64) ALIGNED);                        ❶

/* Clear event flag 5 */
        STS$VALUE = SYS$CLREF(5);
        IF ^STS$SUCCESS THEN RETURN(ST$VALUE);     ❷

/* Set the timer for 10 seconds */
        STS$VALUE = SYS$SETIMR(5,GETBINTIM('0 00:00:10'),,);  ❸
        IF ^STS$SUCCESS THEN RETURN(STS$VALUE);

/* Wait for event flag 5 */
        STS$VALUE = SYS$WAITFR(5);
        IF ^STS$SUCCESS THEN RETURN (STS$VALUE);   ❹

        PUT SKIP LIST('Timer up!');

RETURN(1);
END;
```

❶ This procedure uses the GETBINTIM function (see Section B.4.2.1) to convert an ASCII time value to the system's 64-bit format.

❷ The SYS$CLREF system services ensures that event flag 5 is clear before the Set Timer system service is invoked.

❸ The procedure invokes SYS$SETIMR, specifying by its first argument that SYS$SETIMR should set event flag 5 when the time expires. The argument list contains a reference to GETBINTIM, which returns the system time value for 10 seconds.

❹ The procedure uses the Wait for Event Flag (SYS$WAITFR) system service to wait for the event flag specified in the call to SYS$SETIMR. When the flag is set, the procedure displays a message and exits.

Appendix C
ASCII Character Set

The American Standard Code for Information Interchange (ASCII) is a set of eight-bit numeric values that represent the alphabet, numerals, punctuation, and symbols used in text and in communications protocol. This is the ASCII character set. Table C–1 lists the set and its numeric values.

Table C-1: ASCII Character Set

ASCII Decimal Number	Character	Meaning
0	NUL	Null
1	SOH	Start of heading
2	STX	Start of text
3	ETX	End of text
4	EOT	End of transmission
5	ENQ	Enquiry
6	ACK	Acknowledgement
7	BEL	Bell
8	BS	Backspace
9	HT	Horizontal tab
10	LF	Line feed
11	VT	Vertical tab
12	FF	Form feed
13	CR	Carriage return
14	SO	Shift out
15	SI	Shift in
16	DLE	Data link escape
17	DC1	Device control 1
18	DC2	Device control 2
19	DC3	Device control 3
20	DC4	Device control 4
21	NAK	Negative acknowledgement
22	SYN	Synchronous idle
23	ETB	End of transmission block
24	CAN	Cancel
25	EM	End of medium
26	SUB	Substitute
27	ESC	Escape
28	FS	File separator
29	GS	Group separator
30	RS	Record separator
31	US	Unit separator
32	SP	Space or blank
33	!	Exclamation mark
34	"	Quotation mark
35	#	Number sign
36	$	Dollar sign
37	%	Percent sign
38	&	Ampersand
39	'	Apostrophe
40	(	Left parenthesis
41	)	Right parenthesis
42	*	Asterisk
43	+	Plus sign
44	,	Comma
45	-	Minus sign or hyphen
46	.	Period or decimal point
47	/	Slash

Table C-1: ASCII Character Set (Cont.)

ASCII Decimal Number	Character	Meaning
48	0	Zero
49	1	One
50	2	Two
51	3	Three
52	4	Four
53	5	Five
54	6	Six
55	7	Seven
56	8	Eight
57	9	Nine
58	:	Colon
59	;	Semicolon
60	<	Left angle bracket
61	=	Equal sign
62	>	Right angle bracket
63	?	Question mark
64	@	At sign
65	A	Uppercase A
66	B	Uppercase B
67	C	Uppercase C
68	D	Uppercase D
69	E	Uppercase E
70	F	Uppercase F
71	G	Uppercase G
72	H	Uppercase H
73	I	Uppercase I
74	J	Uppercase J
75	K	Uppercase K
76	L	Uppercase L
77	M	Uppercase M
78	N	Uppercase N
79	O	Uppercase O
80	P	Uppercase P
81	Q	Uppercase Q
82	R	Uppercase R
83	S	Uppercase S
84	T	Uppercase T
85	U	Uppercase U
86	V	Uppercase V
87	W	Uppercase W
88	X	Uppercase X
89	Y	Uppercase Y
90	Z	Uppercase Z
91	[	Left square bracket
92	\	Back slash
93	]	Right square bracket
94	^ or ↑	Circumflex or up arrow
95	← or _	Back arrow or underscore

Table C-1: ASCII Character Set (Cont.)

<table>
<tr><th>ASCII Decimal Number</th><th>Character</th><th>Meaning</th></tr>
<tr><td>96</td><td>`</td><td>Grave accent</td></tr>
<tr><td>97</td><td>a</td><td>Lowercase a</td></tr>
<tr><td>98</td><td>b</td><td>Lowercase b</td></tr>
<tr><td>99</td><td>c</td><td>Lowercase c</td></tr>
<tr><td>100</td><td>d</td><td>Lowercase d</td></tr>
<tr><td>101</td><td>e</td><td>Lowercase e</td></tr>
<tr><td>102</td><td>f</td><td>Lowercase f</td></tr>
<tr><td>103</td><td>g</td><td>Lowercase g</td></tr>
<tr><td>104</td><td>h</td><td>Lowercase h</td></tr>
<tr><td>105</td><td>i</td><td>Lowercase i</td></tr>
<tr><td>106</td><td>j</td><td>Lowercase j</td></tr>
<tr><td>107</td><td>k</td><td>Lowercase k</td></tr>
<tr><td>108</td><td>l</td><td>Lowercase l</td></tr>
<tr><td>109</td><td>m</td><td>Lowercase m</td></tr>
<tr><td>110</td><td>n</td><td>Lowercase n</td></tr>
<tr><td>111</td><td>o</td><td>Lowercase o</td></tr>
<tr><td>112</td><td>p</td><td>Lowercase p</td></tr>
<tr><td>113</td><td>q</td><td>Lowercase q</td></tr>
<tr><td>114</td><td>r</td><td>Lowercase r</td></tr>
<tr><td>115</td><td>s</td><td>Lowercase s</td></tr>
<tr><td>116</td><td>t</td><td>Lowercase t</td></tr>
<tr><td>117</td><td>u</td><td>Lowercase u</td></tr>
<tr><td>118</td><td>v</td><td>Lowercase v</td></tr>
<tr><td>119</td><td>w</td><td>Lowercase w</td></tr>
<tr><td>120</td><td>x</td><td>Lowercase x</td></tr>
<tr><td>121</td><td>y</td><td>Lowercase y</td></tr>
<tr><td>122</td><td>z</td><td>Lowercase z</td></tr>
<tr><td>123</td><td>{</td><td>Left brace</td></tr>
<tr><td>124</td><td>|</td><td>Vertical line</td></tr>
<tr><td>125</td><td>}</td><td>Right brace</td></tr>
<tr><td>126</td><td>~</td><td>Tilde</td></tr>
<tr><td>127</td><td>DEL</td><td>Delete</td></tr>
</table>

INDEX

A format item, 298
ABS built-in function, 338
ABS preprocessor built-in function, 399
Absolute value
  computing, 338
Access modes
  record files, 302
ACOS built-in function, 338
%ACTIVATE statement, 380, 382
ADD built-in function, 338
Addition operator (+), 174
ADDR built-in function, 140, 339
  passing pointer value, 208
  using, 133
Addressable variable, 122
Aggregates, 144
  arrays, 144
  structures, 149
ALIGNED attribute, 118
Alignment
  of bit strings, 119
ALLOCATE command, 12
  allocating tape drive, 319
  establishing logical name, 253
ALLOCATE statement, 130, 139
Allocation
  device, 12
    determining status, 270
  disk file space
    extending, 268
ALLOCATION built-in function, 140, 339
Alternate keys, 327
  accessing records by, 332
AND operator (), 175
ANSI magnetic tape labels, 320
ANY attribute, 207
ANYCONDITION condition, 232
Apostrophes
  in character strings, 113
  in edit-directed I/O, 297
  with GET LIST, 283
APPEND
  ENVIRONMENT option, 256
    determining if set, 265
    example, 318
Arc cosine
  computing, 338
Arc sine
  computing, 339
Arc tangent
  computing
    in degrees, 340
    in radians, 340
AREA attribute, 127
Areas, 127
  allocating variables within, 127
  in assignment statement, 128, 172
Argument list, 198
  for exception condition, 356
  maximum number of arguments, 198
  null, 195
  passing to function, 195
  specifying in CALL statement, 193
  variable-length, 413
Arguments, 198
  aggregate, 201
  arrays, 199
  built-in functions, 337
    restrictions, 337
  character strings, 200
  conversion, 202
  dummy argument, 201
  for system services, 411
    data types, 412 to 413
    methods of passing, 411
    omitting, 413
  list, 198
  matching with parameter, 201
  maximum number in list, 198
  passing, 200
    by descriptor, 208, 346
    by immediate value, 206
    by reference, 207

Arguments,
  passing (Cont.)
    conversion of values, 402
    forcing passing by descriptor, 209
    to function, 195
    to PL/I procedure, 201
    to subroutines and functions, 189
  relationship to parameters, 198
  specifying pointer values, 208
  structures, 199
Arithmetic
  operation
    division, 347
  preprocessor
    built-in functions, 399
Arithmetic data, 96
  in assignment statement, 172
  specifying precision, 111
Arithmetic operators, 85
Arrays, 144
  assigning values with GET
    statement, 148
  assignment statement, 148
  concatenating with STRING, 363
  connected, 161
  declaring, 144
    as parameters, 199
  dimensions
    determining bounds, 349, 351
    determining extent of, 346
    rules for specifying, 145
  elements
    referring to, 146
  extent of, 145
  handling
    summary of functions, 337
  in GET statements, 279
  initializing, 147
  of structures, 160
    referring to elements, 160
    unconnected arrays, 162
  order of assignment and output, 149
  passing
    to non-PL/I procedures, 209
  passing as arguments, 199, 207
    by descriptor, 208
  specifying in assignment, 172
  subscripts, 146
  unconnected, 161
  with edit-directed I/O, 298
ASCII character
  obtaining integer value, 360
ASCII character set
  obtaining string, 344
  table, 421
ASIN built-in function, 339
Assembly language code
  printing in listing file, 49
ASSIGN command, 10
Assignment, 179
  conversion during, 179
%Assignment statement, 380, 382
Assignment statement, 171
  and unconnected arrays, 161
  conversion during
    arithmetic data, 178
    values, 402
  specifying area variables, 128
  specifying array variables, 148
  structures, 160
Asterisk (*)
  EDT prompt, 19
  in array declaration, 145
Asterisk (*) picture character, 107
At-sign (@) command, 13
ATAN built-in function, 340
ATAND built-in function, 340
ATANH built-in function, 341
Attributes
  alphabetic summary, 89
  default, 164
  default arithmetic, 96
  device, determining, 269
  factor in declaration, 165
  file
    alphabetic summary, 245
    determining, 268
    implied by DELETE, 314
    implied by READ, 304
    implied by REWRITE, 311
    implied by WRITE, 308
    record files, 302
    specifying on OPEN, 244
    stream I/O, 276
  matching parameter and argument, 202
  of structure variables, 151
  specifying in DECLARE statement, 163
Automatic storage class, 123

B format item, 299
  specifying base, 299
B picture character, 110
BACK SPACE key
  use in EDT, 37
BASED attribute, 126, 129
  attributes conflicting with, 129
Based variables, 126
  data type matching, 136

Based variables,
  data type matching (Cont.)
    left-to-right equivalence, 137
    overlay defining, 137
  declaring, 129
  example, 135
  freeing storage, 131
  nonmatching reference, 138
  obtaining storage for, 130
  offset within area, 128
  REFER option, 153
  referring to, 131
  using READ statement, 132
BATCH
  ENVIRONMENT option, 256
    determining if set, 265
    specifying on CLOSE, 250
Batch jobs
  compiler errors during, 56
  linker behavior in, 63
  submitting, 13
Begin blocks, 82, 216
  effect of RETURN statement, 196
  in ON-unit, 231
  terminating, 217 to 218
BEGIN statement, 216
Binary
  fixed-point data, 97
  floating-point data, 100
BINARY attribute, 97
  in floating-point declarations, 101
BINARY built-in function, 341
BIT attribute, 118
BIT built-in function, 342
Bit strings, 115
  acquired by GET LIST, 282
  alignment, 119
  as targets in assignment statements, 173
  constants, 117
    hexadecimal, 117
    maximum length, 117
    octal, 117
    specifying base, 117
  converting from bit strings, 119
    to arithmetic, 179, 403
    to character, 179, 408
  converting to bit strings, 342, 404
  declaring variables, 118
  derived type and precision of, 178
  determining length, 351
  length
    maximum, 116
    specifying, 118
  locating substring, 349
Bit strings, (Cont.)
  modifying
    SUBSTR pseudovariable, 185
  operators for, 175
  passing as arguments
    by reference, 207
    by value, 207
  returned by UNSPEC, 368
  storage in memory, 116
  unaligned
    passing as arguments, 208
    restrictions on use, 119
  variables, 118
  with edit-directed I/O, 297, 299
Block activation, 82
  procedure invocation, 189
Block I/O
  space blocks, 272
block size
  determining, 265
BLOCK_BOUNDARY_FORMAT
  ENVIRONMENT option, 256
    determining if set, 265
BLOCK_IO
  ENVIRONMENT option, 256
    determining if set, 265
BLOCK_SIZE
  ENVIRONMENT option, 256
Blocks, 82
  Begin block, 216
  terminating, 217
BOOL built-in function, 342
Boolean
  operation
    define with BOOL, 342
  value, 115
Bound pair
  array, 144
Bounds
  of array dimensions
    determining, 349, 351
    rules, 145
Bucket size
  determining, 265
Bucket splitting, 327
BUCKET_SIZE
  ENVIRONMENT option, 256
Buffers
  argument
    passing by reference, 412
  EDT text, 27
  file system
    flushing, 271
  type-ahead
    purging, 280

Built-in functions, 334
  arguments to
    restrictions, 337
  arithmetic, 335
  array-handling, 337
  condition-handling, 336
  conditions in, 337
  conversion, 178, 336
  file control, 337
  mathematical, 335
  miscellaneous, 337
  storage, 337
  string-handling, 336
  summary, 334
  timekeeping, 337
Built-in subroutines
  DISPLAY, 263
  EXTEND, 268
  file-handling, 263
  FLUSH, 271
  NEXT_VOLUME, 271
  RESIGNAL, 242
    using, 232
  REWIND, 272
  SPACEBLOCK, 272
BUILTIN attribute, 195
BY option
  DO statement, 214
BYTE built-in function, 343
BYTE preprocessor built-in
    function, 399

CALL statement, 193
  calling non-PL/I procedures, 206
    passing character strings, 209
  to invoke a procedure, 83
CANCEL_CONTROL_O option
  PUT statement, 287
Carriage control
  determining, 266
  FTN, 269
    determining if file has, 269
CARRIAGE_RETURN_
    FORMAT
  ENVIRONMENT option, 256
    determining if set, 265
CDD, 372, 374, 386
  data types, 375
  including definitions in listing, 50
CEIL built-in function, 343
Cell
  in relative file, 321
CHANGE command
  EDT, 20
CHANGE command, (Cont.)
  using, 35
Channel number
  specifying as argument, 412 to 413
CHARACTER attribute, 114
CHARACTER built-in function, 344
Character editing mode, 34
  entering, 35
  exiting, 35
  obtaining help, 21
Character set
  ASCII, 421
    obtaining string, 344
Character strings, 113
  acquired by GET LIST, 282
  and stream I/O, 296
  as procedure arguments
    for system services, 412
    passing by descriptor, 412
  as targets in assignment
    statements, 173
  comparing with VERIFY, 370
  constants, 113
    as arguments, 416
    continuing on more than one line,
        86
  converting from character strings
    to arithmetic, 179, 403
    to bit, 179, 404
  converting to character strings, 344,
      405
  declaring, 114
    as parameters, 200
  derived type and precision of, 178
  fixed-length, 114
  form of PUT LIST output, 290
  keys in indexed files, 331
  length
    determining, 351
    specifying, 114
  locating substring, 349
  modifying
    SUBSTR pseudovariable, 185
  passing as arguments, 200
    by descriptor, 209
  variables, 114
    declaring, 114
  varying-length, 115
  with edit-directed I/O, 297 to 298
Characters
  ASCII
    table, 421
  picture, 106
  substituting with

Characters
  substituting with, (Cont.)
    TRANSLATE, 366
  used for punctuation in PL/I, 84
/CHECK qualifier, 47
Circumflex (^)
  prefix operator, 174
CLEAR command
  EDT, 20
CLOSE statement, 250
COLLATE built-in function, 344
Colon
  in TITLE option, 252
COLUMN format item, 299
Column number
  determining current, 269
COM file type, 13
Comma (,) picture character, 110
Command files
  EDT, 43
    unexpected effect of, 23
Command procedures, 13
  data in, 15
  executing, 13
  for program development, 13
  handling errors, 14
  login command file, 16
  passing parameters to, 14
/COMMAND qualifier, 44
Commands
  DCL
    hints for entering, 4
    interrupting execution of, 4
    obtaining information on, 4
    shortening, 4
    specifying qualifiers, 1
  EDT
    shortening, 19
    summary of, 19
    using, 19
  file maintainence, 11
  program development, 1
Comments, 92
  rules for entering, 92
Common Data Dictionary, 372, 374, 386
Comparison operators, 85, 174
Compile time facilties, 372
Compiler
  controlling optimization, 49
  diagnostic messages
    format, 55
  input and output files, 53
  listing, 49
Compiler, (Cont.)
  listing options, 48, 50
  options, 47
Compiler functions, 45
Completion
  ON-unit, 232
Concatenation
  COPY built-in function, 344
  operator for, 175
  required operands, 175
Concatenation operator (|| or !!), 175
Condition handlers
  default PL/I, 229
Condition handling
  functions for
    summary, 336
  ON statement, 227
Condition values
  file errors, 273
Conditions
  ANYCONDITION, 232
  decimal overflow, 236
  ENDFILE, 234
  ENDPAGE, 234
  ERROR, 235
  FINISH, 236
  FIXEDOVERFLOW, 236
  in built-in functions, 337
  integer overflow, 236
  KEY, 237
  OVERFLOW, 238
  resignaling, 242
  run-time, 67
  signaling, 242
  UNDEFINEDFILE, 238
  UNDERFLOW, 239
  VAXCONDITION, 239
  ZERODIVIDE, 240
Connected array, 161
Constants
  bit string, 117
  character string, 113
  entry, 204
    declaring implicitly, 190, 192
    external, 204
  file, 244
  fixed-point decimal, 98
  floating-point, 100
  in argument list, 201
  integer, 98
  label, 222
  label array, 223
Containment, 168

CONTIGUOUS
  ENVIRONMENT option, 256
    determining if set, 265
CONTIGUOUS_BEST_TRY
  ENVIRONMENT option, 256
    determining if set, 265
CONTINUE command, 4, 69
CONTROLLED attribute, 138
Controlled DO statement, 213
Controlled variable, 138
  obtaining storage for, 130
Conversion, 180
  arithmetic to arithmetic, 402
  arithmetic to bit string, 404
  arithmetic to character string, 405
  ASCII to integer, 360
  bit string to arithmetic, 403
  bit string to character string, 408
  character string to arithmetic, 403
  character string to bit string, 404
  data
    rules for, 402
  fixed-point to character string, 405
  floating-point to character string, 406
  integer to ASCII, 343
  of argument, 202
  of operands, 177
  offset to pointer, 408
  picture to arithmetic, 403
  picture to bit string, 404
  picture to character string, 407
  pointer to offset, 408
  summary of functions, 336
  to arithmetic, 402
  to binary, 341
  to bit string, 342, 404
  to character string, 344, 405
  to decimal, 346
  to fixed point, 348
  to floating point, 349
  to picture, 408
COPY built-in function, 344
COPY command
  DCL, 12
  EDT, 20
    using, 31
COPY preprocessor built-in function, 399
COS built-in function, 345
COSD built-in function, 345
COSH built-in function, 345
Cosine
  computing
    from degree argument, 345
Cosine
  computing, (Cont.)
    from radian argument, 345
  computing hyperbolic, 345
CREATE command, 11
/CREATE qualifier, 74
CREATE/DIRECTORY
    command, 8, 11
Creation date of file
  determining, 265
CREATION_DATE
  ENVIRONMENT option, 256
    example, 418
Credit (CR) picture character, 111
Cross-reference listing file
  printing
    cross references, 47
/CROSS_REFERENCE qualifier, 47
CTRL U
  use in EDT, 39
CTRL Y
  interrupting DCL commands, 4
CTRL/C
  effect on PL/I program, 69
  handling routine, 70
CTRL/O
  disabling, 288
CTRL/Y
  effect on PL/I program, 69
Current line, 25
  as default range, 25
    DELETE command, 30
    INSERT command, 29
  changing location of, 28
  specifying, 25
Current record, 301
CURRENT_POSITION
  ENVIRONMENT option, 256
    determining if set, 265
Cursor
  positioning in EDT, 35
    direction of movement, 37
    to a string, 37

D floating-point format, 48 to 49
DAT file type, 255
  usage, 252
Data, 88
  arithmetic
    converting from other types, 402
    converting to bit string, 404
    converting to character string, 405
  conversion, 180
    rules for, 402
  in command procedures, 15

Data type attributes
  alphabetic summary, 89
Data types, 95
  arguments
    passed by descriptor, 208
    passed by immediate value, 206
    passed by reference, 207
  arithmetic, 96
    converting to nonarithmetic, 178
    default attributes, 96
    default precision, 112
    fixed-point binary, 97
    precision of, 111
  behavior in assignment
    general rules, 172
  bit string, 115
  character string, 113
  computational, 95
  conversion between, 177
  derived, 177
  entry, 204
  file, 243
  fixed-point binary, 97
  fixed-point decimal, 98
  floating-point, 100
  for CDD declarations, 375
  for keys in indexed files, 330
  for system service arguments, 412 to 413
  nonarithmetic
    converting to arithmetic, 178
  noncomputational, 95
  picture, 103
  pointer, 127
  summary, 95
DATE built-in function, 345
DATE preprocessor built-in function, 399
Day of month
  obtaining current, 345
%DEACTIVATE statement, 380, 384
DEASSIGN command, 10
Debit (DB) picture character, 111
DEBUG command, 69
/DEBUG qualifier
  PLI, 48
Debugger
  compile-time options, 48
  invoking at run time, 69
DECIMAL attribute, 99
  in floating-point declarations, 101
DECIMAL built-in function, 346
Decimal data
  fixed-point data, 98
  floating overflow, 238
Decimal data, (Cont.)
  floating underflow, 239
  floating-point data, 100
Decimal place
  in picture, 107
Declarations, 163
  array, 144
  factored, 165
  initializing variables in, 166
  multiple, 164
  of variables with same attributes, 165
  scope of, 168
  simple, 164
  structure, 150
    level numbers, 149
%DECLARE, 385
%DECLARE statement, 381
DECLARE statement, 163
  declaring files, 243
Default attributes
  arithmetic, 96
Default file specifications, 5
  at open, 254
  changing, 8
  PLI command, 47
  used by linker, 62
Default libraries
  INCLUDE modules, 55
    PLI$LIBRARY, 55
    PLISYSDEF, 55
  object module, 64
    LNK$LIBRARY, 64
    STARLET.OLB, 65
DEFAULT_FILE_NAME
  ENVIRONMENT option, 256
Defaults
  directory, 8
    changing, 8
    EDT, 33
  file specification elements, 6
  line size, 293
  page size, 295
  program development commands, 5
DEFERRED_WRITE
  ENVIRONMENT option, 256
    determining if set, 265
DEFINE command
  DCL, 10
    creating logical names, 9
    DEFINE/USER, 10
    defining program I/O files, 252
  DEFINE/USER
    in command procedure, 15

DEFINE command, (Cont.)
  EDT, 20
    DEFINE KEY, 43
DEFINED attribute, 141
DELETE
  ENVIRONMENT option, 256
    determining if set, 265
    specifying on CLOSE, 250
DELETE command
  DCL, 12
  EDT, 20
    using, 30
DELETE key
  use in EDT, 39
/DELETE qualifier, 74
DELETE statement, 314
  file position following, 304
Derived type, 177
  of bit and character strings, 178
Descriptor
  argument passing, 208
  data types created for, 208
DESCRIPTOR built-in function, 346
  specifying in argument, 208
  using, 209
Device attributes
  returned by DISPLAY, 269
Devices
  default, 254
  default line size, 277
devices
  specifying
    rules, 6
Diagnostic message
  user-generated, 57
DICTIONARY statement, 374
%DICTIONARY statement, 374, 381, 386
DIMENSION built-in function, 346
Dimensions
  array of structures
    rules, 161
  rules for specifying, 145
DIRECT attribute
  determining if file has, 269
  specifying on OPEN, 245
Directories
  changing default, 8
  default, 254
  EDT default, 33
  specifying, 8
  SYS$LIBRARY, 65
DIRECTORY command, 11
directory specifications
  rules, 6
Disk files
  extending allocation, 268
Disks
  default line size, 293
DISPLAY built-in subroutine, 263
  device information, 269
  ENVIRONMENT information, 265
  file attribute information, 269
DIVIDE built-in function, 347
Division
  control precision, 347
  ZERODIVIDE condition, 240
Division operator (/), 174
%DO, 387
DO option
  GET statement, 279
  PUT statement, 285
%DO statement, 381
DO statement, 210
  controlled DO, 213
  DO REPEAT, 215
  DO UNTIL, 212
  DO WHILE, 211
  simple, 210
DO-groups
  terminating, 218
Documentation
  program, 92
Dollar ($) picture character, 109
Double-precision floating point
  range of precision, 102
Drifting picture characters, 109
Dummy argument, 201
  forcing creation of, 202
Duplicate keys
  testing for errors, 274

E format item, 299
EDIT command, 3, 11
  in command procedure, 15
EDIT option
  GET statement, 280
  PUT statement, 288
EDIT/EDT command, 22
  parameter, 22
  response, 23
    unexpected, 23
  to create new file, 24
  to revise existing file, 24
  with /RECOVER qualifier, 40
EDIT/FDL
  examples, 329
Editing session
  recovering lost, 40

Editors
  invoking, 3, 11
    in command procedure, 15
EDT
  help facilities
    line mode, 21
EDT, 18
  computer-assisted course, 18
  help facilities
    character mode, 21
  invoking, 22
  operating modes, 18
  programming aids, 41
  summary of commands, 19
  summary of features, 19
  terminating, 23
EDTINI.EDT, 43
Elements
  array, 145
    referring to, 146
Embedded preprocessor, 376
  statements, 380
Empty argument list, 195
Encoded-sign picture characters, 108
%END statement, 381, 387
END statement, 217
  in main procedure, 67
  terminating subroutine or
    function, 190
End-of-tape
  on volume, 320
End-of-volume switching, 320
ENDFILE condition, 234
  signaled, 273
    by GET LIST, 283
    READ statement, 306
    stream files, 293
ENDPAGE condition, 234
  default PL/I action, 229
  signaled, 273, 277, 287, 294 to 295
ENTRY
  statement, 191
Entry
  constants, 204
  data type, 204
  variables, 206
ENTRY attribute, 204
  declaring non-PL/I procedures,
    206
Entry constants, 204
  declaring implicitly, 190, 192
  external, 204
    declaring, 204
Entry data
  in assignment statements, 173
Entry points, 45
  alternate, 191
  invoking, 189
  main, 60, 67
  multiple, 192
  primary
    identifying, 191
  procedure, 188
    identifying, 190
  specifying attributes of
    return value, 196
ENTRY statement
  RETURNS option, 196
Entry variables, 206
ENVIRONMENT attribute
  CLOSE options, 250
  specifying on OPEN, 245
ENVIRONMENT options
  obtaining information, 264
  summary, 255
Error (severity)
  meaning to compiler, 56
  meaning to linker, 60
ERROR condition, 235
  default PL/I action, 229
  determine error status value, 356
  signaled
    by default handler, 229
    converison of character strings,
      403
    conversion of values, 408
    for file errors, 273
    GET EDIT, 297
    GET LIST, 283
    in assignment to pictured
      variable, 105
    READ statement, 305
    stream I/O on character strings,
      296
    WRITE statement, 309
Error handling
  in command procedure, 14
  of file-related error, 357
  ON conditions, 227
  ONCODE built-in function, 356
Error message, 388
ERROR preprocessor built-in
  function, 399 to 400
%ERROR statement, 381, 388
/ERROR_LIMIT qualifier, 48
Errors
  arithmetic operations
    divide by zero, 240
  at run time, 68
    conversion, 179

Errors, (Cont.)
  compiler
    effect on linker, 60
    implicit conversion, 178, 180
    message format, 55
  displaying system messages, 70
  file
    default handling, 275
    error handler, 274
    handling opening error, 238
  handling, 227
    file errors, 273
    VAX-specific conditions, 239
  indexed sequential files, 333
  linking, 60
  relative files, 326
  syntax
    detected by compiler, 57
Evaluation
  of expressions, 176
Event flags
  clearing, 418
  specifying as argument, 412
  waiting for, 418
  with a timer, 418
Exclusive OR, 342
EXE file type, 62, 67
Executable images
  creating, 59
/EXECUTABLE qualifier, 63
Execute Procedure (@) command, 13
EXIT command
  DCL, 69
  EDT, 20
    using, 23
EXP built-in function, 347
Expiration date of file
  determining, 265
EXPIRATION_DATE
  ENVIRONMENT option, 257
    example, 418
Exponent
  floating-point data, 100
Exponentiation
  order of evaluation, 177
Exponentiation operator (**), 174
Expressions, 175
  area variables in, 128
  conversion
    of operands, 177
  derived type, 177
  evaluation, 176
    order of, 176
  in argument list, 201
  offset variables in, 129
Expressions, (Cont.)
  restricted integer, 145
  using as subscripts, 146
EXTEND built-in subroutine, 268
Extension size
  determining, 265
EXTENSION_SIZE
  ENVIRONMENT option, 257
Extent
  area, 128
  array, 145
  array dimension
    determining, 346
  static variables, 123
  structure members, 151
EXTERNAL attribute, 124
External procedures, 83, 202
External storage class, 124
/EXTRACT qualifier, 74

F format item, 299
Facility name, 56
Factored declarations, 165
FAST_DELETE option, 315
  DELETE statement, 315
Fatal (severity)
  meaning to compiler, 56
  meaning to linker, 60
Fatal message, 388
%FATAL statement, 381, 388
File
  source
    %INCLUDE text, 390
FILE attribute, 243
File attributes
  determining current, 269
File constants
  associating with VAX/VMS file, 251
File identification
  determining, 265
File information
  displaying values, 263
file names
  rules, 7
FILE option
  format, 244
  GET statement, 279
  PUT statement, 286
File organization
  determining, 269
File size
  determining, 265
File specifications, 5
  completing, 252, 254
  defaults, 5

File specifications
  defaults, (Cont.)
    file opening, 254
    GET statement, 279
    in TITLE option, 252
    PUT statement, 286
  expanded
    determining, 269
  for error, 357
  GET statement, 279
  invalid, 252
  logical names, 8
  PUT statement, 286
  relating to file constant, 251
  specifying in OPEN, 247
  TITLE option, 251
file specifications
  rules, 6
File types
  COM, 13
  DAT, 255
    usage, 252
  default
    used by commands, 5
    used by PL/I, 255
  EXE, 62, 67
  LIS, 49
  MAP, 63
  OBJ, 49, 60, 62, 79
  OLB, 62, 73, 79
  OPT, 62
  PLI, 47
  TLB, 47, 52, 76
  used by LIBRARY command, 79
  used by LINK command, 62
file types
  rules, 7
File version numbers
  default, 255
file version numbers
  rules, 7
FILE_ID
  ENVIRONMENT option, 257
FILE_ID_TO
  ENVIRONMENT option, 257
FILE_SIZE
  ENVIRONMENT option, 257
Files
  attributes
    implied, 248
    merging at open, 248
    specifying on OPEN, 245
  built-in subroutines, 263
  closing, 250
  commands for maintaining, 11
Files, (Cont.)
  compiler input and output, 53
  constants, 244
  creating
    with EDT, 24
    with OPEN, 249
  creation date
    example, 418
  data type, 243
  declaring, 243
  deleting records, 314
  determining end, 234
  displaying information, 263
  EDT input from, 33
  EDT output to, 33
  EDT protection against
    modification, 40
  error conditions, 273
  errors
    error handler, 274
  expiration date
    example, 418
  functions for controlling
    summary, 337
  indexed sequential, 326, 329
    creating, 329
    error handler example, 274
    handling errors, 333
    organization, 327
    reading sequentially, 331
    updating, 333
    using, 330
  key error, 237
  linker input, 62
  linker options, 62
  linker output, 62
  magnetic tapes, 319
  opening, 247
    effects of, 247
    for input, 246
    for output, 247
    OPEN statement, 244
    positioning, 250
    UNDEFINEDFILE condition, 238
  positioning at beginning, 272
  print, 294
    changing page number, 183
    determining current page number, 357
  process permanent, 253
  record, 301
    access modes, 302
    attributes for, 302
    position information, 301
  relative, 321

Files, (Cont.)
creating, 321
handling errors, 326
organization, 321
reading and writing, 325
reading sequentially, 325
updating, 325
using, 323
revising, 24
deleting text, 30
inserting new text, 29
moving text, 31
sequential, 318
creating, 318
specifying, 5
specifying line size, 276
specifying page size, 277
statements for controlling, 243
stream, 277
updating record, 311
variables, 244
writing
to spooled devices, 252
FILL command
EDT, 20
FIND command
EDT, 20
using, 29
FINISH condition, 67, 236
at image exit, 69
signaled
STOP statement, 190, 226
FIXED attribute, 97, 99
FIXED built-in function, 348
Fixed control area
determining size, 265
reading, 316
writing or rewriting, 316
Fixed-point data
binary, 97
interpreting as bit string, 116
range of precision, 97
range of values, 98
decimal, 98
constant, 98
declaring variables, 99
precision, 99
range of precision, 99
scale factor, 99
overflow condition, 236
with edit-directed I/O, 299
FIXED_CONTROL_FROM
option, 316
REWRITE statement, 312
WRITE statement, 309
FIXED_CONTROL_SIZE
ENVIRONMENT option, 257
FIXED_CONTROL_SIZE_TO
ENVIRONMENT option, 257
FIXED_CONTROL_TO option,
316
READ statement, 306
FIXED_LENGTH_RECORDS
ENVIRONMENT option, 257
determining if set, 266
FIXEDOVERFLOW condition, 236
signaled
assignment to pictured
variable, 105, 107
conversion of bit strings, 403
conversion of character strings,
403
conversion of values, 402, 408
exceeding maximum integer
value, 98
FLOAT attribute, 101
FLOAT built-in function, 349
Floating-point data, 100
constants, 100
declaring variables, 101
default precision, 103
OVERFLOW condition, 238
precision, 101
range of values, 102
selecting default format, 48
UNDERFLOW condition, 239
using in expressions, 101
with edit-directed I/O, 299
FLOOR built-in function, 349
FLUSH built-in subroutine, 271
Format
of source program, 41, 92
Format items, 296
control
positioning with, 293
data, 300
iteration factor, 297
order of execution, 300
repetition of, 297
summary, 298
Format specifications, 297
FORMAT statement, 292
label restriction, 223
Format-specification lists
remote, 292
Fractional digits
specifying, 97, 99
FREE statement, 131, 139
FROM option
REWRITE statement, 312

FROM option, (Cont.)
WRITE statement, 308
FTN carriage control, 269
Functions, 188, 194
built-in, 334
summary, 334
external, 202
invoking procedure with, 83
invoking with no arguments, 195
references to, 194
RETURN statement, 195
returning value from, 195
specifying attributes of return value, 196
terminating, 189
user-written
requirements, 194

G floating-point format
range of precision, 102
selecting at compile time, 48
/G_FLOAT qualifier, 48
G_FLOAT support, 102
GET statement, 278
assigning values to array elements, 148
conversion of values, 179, 402
default file attributes, 279
default file title, 254
DO option, 279
FILE option, 279
forms, 278
GET EDIT, 280
GET LIST, 282
GET SKIP, 284
input target, 279
options, 280
SKIP option, 280
STRING option, 279
Global symbol table, 78 to 79
Global symbols
declaring, 414
referencing in ON-unit, 274
GLOBALREF attribute, 414 to 415
GOTO command
using in command procedures, 14
%GOTO statement, 381, 388
GOTO statement, 222
nonlocal GOTO, 190, 222
terminating subroutine or function, 190
Group logical name table, 9
Group number
of file's owner
determining, 266

GROUP_PROTECTION
ENVIRONMENT option, 257
determining current value, 266
Groups
terminating, 217
GRPNAM user privilege, 9

H floating-point format
range of precision, 102
H_FLOAT support, 102
HBOUND built-in function, 349
HELP command
EDT
using, 21
HELP command
DCL, 4
EDT, 20
HIGH built-in function, 349

I picture character, 108
IDENT option
PROCEDURE statement, 191
Identifiers, 86
associating with variables, 88
rules for forming, 86
IF command
using in command procedures, 14
%IF statement, 381, 389
IF statement, 218
nesting, 219
IGNORE_LINE_MARKS
ENVIRONMENT option, 258
determining if set, 266
Image exit, 67
Image files
creating, 59, 62
specifying name for, 63
IMAGELIB.OLB, 65
%INCLUDE
rules for file specifications, 390
INCLUDE command
default directory for, 34
EDT, 20
using, 33
INCLUDE files, 53, 77, 93
extracting from library, 74
including in listing, 50
libraries, 54, 77
correcting, 77
default, 55
specifying in PLI command, 46, 52
nesting level, 93
temporary defaults, 65

/INCLUDE qualifier (LINK command)
when to specify, 62
%INCLUDE statement, 93, 381, 390
for files, 94
for text modules, 94
using, 53
INDEX built-in function, 349
Index number
determining current, 266
reset by WRITE statement, 330
specifying on I/O statements, 316
INDEX preprocessor built-in function, 399
INDEX_NUMBER
ENVIRONMENT option, 258, 332
INDEX_NUMBER option, 316
DELETE statement, 315
READ statement, 306
using, 332
REWRITE statement, 312
INDEXED
ENVIRONMENT option, 258
determining if set, 266
Indexed sequential files, 326, 329
creating, 329
determining if file is indexed, 266
examples, 330
handling errors, 333
key, 327
error handling, 237
key data types, 330
ONKEY built-in function, 357
organization, 327
positioning at beginning of index, 272
reading sequentially, 331
specifying type of key match, 317
speeding up record deletion, 315
updating, 333
using, 330
Infix operators, 174
INFORM preprocessor built-in function, 399 to 400
%INFORM statement, 381, 391
Informational (severity)
meaning to compiler, 56
Informational message, 391
INITIAL attribute, 166
restrictions, 168
INITIAL attribute, (Cont.)
with arrays, 147
with structures, 151
INITIAL_FILL
ENVIRONMENT option, 258
determining if set, 266
INITIALIZE command, 12
initializing magnetic tape, 319
Input
record, 301
READ statement, 304
stream, 278
GET statement, 278
INPUT attribute, 246
determining if file has, 269
specifying on OPEN, 245
Input files
compiler, 53
defining for program I/O, 251
linker, 62
Input/output
area, 128
file specifications, 251
format list, 292
record
statements, 301
record file, 301
statements
DELETE, 314
GET, 278
PUT, 285
READ, 304
REWRITE, 311
WRITE, 308
stream file, 276
INSERT command
EDT, 20
using, 24
to insert in existing file, 29
/INSERT qualifier, 75
Insertion picture characters, 110
INT built-in function, 350
INT pseudovariable, 181
Integer constants
representation, 98
Integer data
overflow condition, 236
Integers
fixed-point binary, 97
fixed-point decimal, 98
interpreting as bit strings, 116
maximum values, 98
Internal procedures, 83
Internal representation
changing with UNSPEC, 186

Internal representation, (Cont.)
  obtaining with UNSPEC, 368
Internal variables, 124
  scope of, 169
Interrupts
  handling with ON statement, 227
INTO option
  READ statement, 304
Iteration factor
  format item, 297
  INITIAL attribute, 147, 167
  picture character, 106

Journal file, 40
  automatic deletion of, 40
  changing default, 40
  default directory for, 33
  disabling, 40
  EDT use of, 40
/JOURNAL qualifier, 40

KEY condition, 237
  determining key that caused, 357
  sample ON-unit, 274
  signaled, 273
    DELETE statement, 315
    indexed sequential file operations, 333
    key value too large, 323
    READ statement, 305
    relative file operations, 326
    REWRITE statement, 312
    WRITE statement, 309
KEY option
  DELETE statement, 314
  READ statement, 305
  required with INDEX_NUMBER, 316
  REWRITE statement, 312
  specifying for relative file, 325
KEYED attribute
  creating relative file, 321
  determining if file has, 269
  specifying on OPEN, 245
KEYFROM option
  WRITE statement, 309
Keypad
  redefining keys of, 42
    to insert text, 42
  use of in EDT, 34
  VT100, 36
  VT52, 36
Keys
  alternate, 327
    accessing records by, 332
  binary, 331
  character-string, 331
  data types
    indexed sequential files, 330
  decimal, 331
  determining number, 269
  handling errors, 274
  handling invalid data type errors, 274
  indexed sequential file, 305
  matching key values
    match greater, 317
    match greater or equal, 317
  relative files, 321
  relative or sequential file, 305
  specifying alternate, 332
  values
    relative files, 321
keys
  values
    indexed sequential files, 327
KEYTO option
  READ statement, 305
Keywords, 84
  recognition from context, 84

LABEL attribute, 224
Label constant, 222
  declaring implicitly, 222
Label data
  in assignment statements, 173
Labels, 84, 222
  array constant, 223
  magnetic tape, 320
  subscripted, 223
  transferring control to, 222
  variables, 224
LBOUND built-in function, 351
LEAVE statement, 224
Left-to-right equivalence
  matching based variables by, 137
Length
  of string
    determining, 351
LENGTH built-in function, 351
  using, 115
LENGTH preprocessor built-in function, 399
Level numbers, 149
  rules for specifying, 150

Libraries
  creating, 72
  default user, 64
    defining at login, 17
  IMAGELIB.OLB, 65
  INCLUDE files, 52, 77
    creating, 74
    default, 55
    search order, 54
    system, 55
  listing contents of, 75
  LNK$LIBRARY, 64
  object module, 63, 78
    creating, 74, 78, 80
    default, 64
    search order, 63
    specifying, 63
    system, 65
  PLI$LIBRARY, 55
  PLISYSDEF.TLB, 55
  replacing modules in, 76
  run-time, 65
  specifying
    in LINK command, 62
    in PLI command, 46, 52
  STARLET.OLB, 65
  SYS$LIBRARY, 55
  system, 65
LIBRARY command, 12, 72
  INCLUDE file libraries, 77
  object module libraries, 78
  parameters, 73
  printing system service module, 411
  qualifiers, 73
/LIBRARY qualifier
  LINK command, 62 to 63
  PLI command, 52, 54
LIKE attribute, 152
  using, 152
Line end character, 86
LINE FEED key
  use in EDT, 39
LINE format item, 299
Line numbers
  determining, 295
  EDT, 25
    disabling and restoring, 24
    specifying, 25
  source file
    assigned by compiler, 57
    in run-time traceback, 69
  specifying
    LINE format item, 299
Line numbers, (Cont.)
  stream file
    determining, 352
  stream files
    determining current, 269
LINE option
  PUT statement, 287, 289
LINE preprocessor built-in function, 399 to 400
Line printer
  printing program listing on, 1
    listing file
      printing, 53
  spooling program output, 252
Line size
  default, 277, 293
  determining current, 269
  specifying, 276
LINENO built-in function, 352
  using, 295
LINESIZE option, 276
  specifying on OPEN, 245
LINK command, 3, 59
  in command procedure, 15
  parameters, 59
  qualifiers, 61
  specifying input files, 62
  specifying output files, 62
Linker, 59
  errors, 60
  functions performed by, 59
  input files, 62
  invoking, 59
  messages, 60
  options, 61
  output files, 62
LIS file type, 49
LIST option
  GET statement, 282
  PUT statement, 290
/LIST qualifier
  LIBRARY, 75
  PLI, 49
%LIST statement, 381, 391
Listing file
  printing
    INCLUDE files, 50
    machine code, 49
  rules for naming, 53
LNK$LIBRARY, 64
  defining at login, 17
Locator qualifier
  (—), 131
LOG built-in function, 352

LOG10 built-in function, 352
LOG2 built-in function, 352
Logarithm
  computing base 10, 352
  computing base 2, 352
  computing natural, 352
Logical name tables, 9
  search order, 9
  searching for LNK$LIBRARY, 64
Logical name translation, 9
  on TITLE option, 252
  SYS$TRNLOG system service, 415
Logical names, 8
  commands to manipulate, 10
  defining, 9
    at login, 17
    with MOUNT command, 319
  deleting, 10
  displaying equivalence, 9
  for program I/O files, 252
  LNK$LIBRARY, 64
  PLI$LIBRARY, 55
  process permanent files, 253
  SYS$COMMAND, 254
  SYS$DISK, 254
  SYS$ERROR, 254
  SYS$INPUT, 254
  SYS$LIBRARY, 55, 65
  SYS$LP_LINES, 295
  SYS$OUTPUT, 254
Logical operations
  NOT, 174
Logical operators, 85, 175
Login command file, 16
LOGIN.COM, 16
LOW built-in function, 352
Lowercase and uppercase letters
  in identifiers, 86

/MACHINE_CODE qualifier, 49
Magnetic tapes, 319
  allocating drive, 253
  default line size, 277
  format, 320
  initializing, 319
  labels, 320
  mounting next volume, 271
  multivolume, 271, 320
  rewinding, 272
  setting expiration date
    example, 418
  version numbers, 319
  volume switching, 320
Mailboxes
  determining if file is a mailbox, 270
MAIN option
  PROCEDURE statement, 191
  program transfer address, 60
Main procedure, 83
  exit handler, 67
  identifying, 191
  return status values, 70
Major structure, 150
  restriction on INITIAL, 152
Mantissa, 100
Map file (linker)
  specifying name for, 63
MAP file type, 63
/MAP qualifier, 63
MATCH_GREATER option, 317
  DELETE statement, 315
  READ statement, 306
  REWRITE statement, 312
MATCH_GREATER_EQUAL option, 317
  DELETE statement, 315
  READ statement, 306
  REWRITE statement, 312
Matching
  based variable references, 136
  parameter and argument, 201
Mathematical functions
  summary, 335
MAX built-in function, 353
MAX preprocessor built-in function, 399
Maximum record number, 323
  determining, 266
Maximum record size, 323
  determining, 266
MAXIMUM_RECORD_NUMBER
  ENVIRONMENT option, 258, 323
MAXIMUM_RECORD_SIZE
  ENVIRONMENT option, 258, 323
Member attributes, 152
Member number
  of file's owner
    determining, 266
Message
  diagnostic, 388, 391, 398
Messages
  after image exit, 70
  compiler

Messages
  compiler, (Cont.)
    format, 55
    implicit conversion, 180
    suppressing warning, 180
  facility name, 56
  identification, 57
  linker, 60
  run-time, 68
    format, 68
  severity
    meaning to compiler, 56
  suppressing compiler, 50
MIN built-in Function, 353
MIN preprocessor built-in
    function, 399
Minor structure, 150
  restriction on INITIAL, 152
Minus (-) picture character, 109
Minus sign (-)
  prefix operator, 174
MOD built-in function, 353
MOD preprocessor built-in
    function, 399
Module name
  assigned by compiler, 45
  in run-time traceback, 68
  object module, 79
  table, 78 to 79
  text module
    specifying name for, 54, 77
/MODULE qualifier, 76
Month
  obtaining current, 345
MOUNT command, 12
  establishing logical name, 253
  mounting magnetic tapes, 319
MOVE command
  EDT, 20
  using, 31
Multiblock count
  determining, 266
MULTIBLOCK_COUNT
  ENVIRONMENT option, 259
Multibuffer count
  determining, 266
MULTIBUFFER_COUNT
  ENVIRONMENT option, 259
Multiple entry points, 192
Multiplication operator (*), 174
MULTIPLY built-in function, 355
Multivolume tape files, 320
  mounting next volume, 271
Names
  declaring, 163
Names, (Cont.)
  rules for identifiers, 86
  scope, 168
Nesting
  IF statement, 219
  %INCLUDE statement, 390
  SELECT-group, 220
Next record, 301
NEXT_VOLUME built-in
    subroutine, 271
  using, 320
Nine (9) picture character, 107
NO_ECHO option
  GET statement, 280
NO_FILTER option
  GET statement, 280
NO_SHARE
  ENVIRONMENT option, 259
    determining if set, 266
Node names
  default, 254
%NOLIST statement, 381, 392
Nonaddressable variable, 122
Nonlocal GOTO, 190, 222
Nonmatching based variable
    reference, 138
NOT operator (^), 174 to 175
/NOWARNINGS qualifier, 180
Null argument list, 195
NULL built-in function, 355
  using, 132
Null pointers
  NULL built-in function, 355
Null record, 283
%NULL statement, 382
Null statement, 226
  as target of ELSE, 219
  in ON-unit, 231
  multiple labeled, 222
  preprocessor, 380
Numbers
  level, 149
  page
    changing, 183

OBJ file type, 49, 60, 62, 79
Object module file
  specifying name for, 49
Object module libraries
  using, 63
Object modules
  creating, 49
  extracting from library, 74
  libraries, 78
    creating, 80

Object modules
libraries, (Cont.)
using, 63
linking, 59
specifying label and version, 191
/OBJECT qualifier
LIBRARY, 75
PLI, 49
OFFSET
attribute, 128
Offset
data type, 128
OFFSET built-in function, 355
Offsets
converting to pointers, 358, 408
obtaining values
OFFSET built-in function, 355
OLB file type, 62, 73, 79
ON conditions, 227
alphabetic summary, 228
ENDFILE, 234
ENDPAGE, 234
ERROR, 235
FINISH, 236
FIXEDOVERFLOW, 236
KEY, 237
OVERFLOW, 238
resignaling, 242
signaling, 242
UNDEFINEDFILE, 238
UNDERFLOW, 239
VAXCONDITION, 239
ZERODIVIDE, 240
ON statement, 227
ON-units
argument list for exception, 356
completion, 232
contents of, 230
default PL/I, 229
establishing, 227
examples, 240
for KEY condition
in indexed file, 274
handle any condition, 232
invalid statements in, 231
multiple statements in, 217
referencing global symbols
examples, 274
restoring default handling, 241
scope, 230
search for, 231
ONARGSLIST built-in function, 356
ONCODE built-in function, 356
using, 273
ONCODE built-in function
using, (Cont.)
ERROR condition, 240
KEY condition, 237
UNDEFINEDFILE condition, 238
values in KEY ON-unit, 274
ONFILE built-in function, 357
using, 273
ENDFILE condition, 234
ENDPAGE condition, 235
KEY condition, 237
UNDEFINEDFILE condition, 238
ONKEY built-in function, 357
ONCODE values, 274
using, 273
KEY condition, 237
OPEN statement, 244
specifying TITLE option, 251
Operands, 175
conversion of, 177
Operations
arithmetic, 96, 174
data type of result, 178
required operands, 175
Boolean
defining, 342
comparison
required operands, 175
concatenation
required operands, 175
logical
required operands, 175
termination of evaluation, 177
relational
required operands, 175
Operators, 174
arithmetic, 174
comparison, 174
infix, 174
logical, 175
precedence, 176
table, 176
prefix, 174
relational, 174
OPT file type, 62
Optimization
compile-time options, 49
linking object modules, 79
/OPTIMIZE qualifier, 49
Options
compiler, 47, 50
ENVIRONMENT
summary, 255

Options, (Cont.)
Linker, 61
OPTIONS (MAIN)
as program transfer address, 60, 67
influence on condition handling, 229
OPTIONS (VARIABLE), 205
in subroutine declaration, 193
in system service declarations, 413
Options file, 62
OPTIONS option
DELETE statement, 315
GET statement, 280
PROCEDURE statement, 191
PUT statement, 287
READ statement, 306
REWRITE statement, 312
WRITE statement, 309
/OPTIONS qualifier, 62
OR
exclusive, 342
OR operator (| or !), 175
OTHERWISE clause, 220
Output
PUT statement, 285
record, 301
REWRITE statement, 311
stream, 285
to line printer, 294
to terminal, 294
WRITE statement, 308
OUTPUT attribute, 247
creating a new file, 318
determining if file has, 269
specifying on OPEN, 245
Output files
EDT, 23
disabling creation of, 40
overriding default, 23
linker, 62
program
defining, 251
/OUTPUT qualifier
EDT, 22
LIBRARY, 76
Overflow
fixed-point data, 236
floating-point data, 238
OVERFLOW condition, 238
signaled
conversion of values, 403
Overlay defining, 143
matching based variables by, 137
Owner of a file
determining, 266
OWNER_GROUP
ENVIRONMENT option, 259
OWNER_MEMBER
ENVIRONMENT option, 259
OWNER_PROTECTION
ENVIRONMENT option, 259

P format item, 300
example, 289
Padding
bit strings, 404
character strings, 405
PAGE format item, 300
Page number
current, 357
determining, 269, 295
resetting, 183, 295
PAGE option
PUT statement, 286, 291
Page size
default, 295
specifying, 277
Page size of PRINT files
determining current, 269
%PAGE statement, 381, 392
PAGENO built-in function, 357
using, 295
PAGENO pseudovariable, 183
using, 295
Pages
handling end-of-page condition, 234
PAGESIZE option, 277
specifying on OPEN, 245
Parameter descriptors, 198
VALUE attribute in, 206
Parameter list
in ENTRY statement, 192
in PROCEDURE statement, 190
Parameters, 198
arrays, 199
character strings, 200
declaring, 198
in ENTRY statement, 192
in PROCEDURE statement, 190
matching with argument, 201
maximum number allowed, 199
passing to command procedure, 14
relationship to arguments, 198
rules for specifying, 198
structures, 199
Parentheses
enclosing procedure argument, 202
in expressions, 177

PC (Program Counter)
  in run-time traceback, 69
Period (.) picture character, 110
Picture, 103
  character, 106
  converting from other types, 408
  converting to arithmetic, 403
  converting to bit string, 404
  converting to character string, 407
  declaring variables, 104
  drifting characters, 109
  extracting value from, 105
  inputting with READ, 369
  insertion characters, 110
  validating, 369
  with edit-directed I/O, 300
PICTURE attribute, 104
Picture characters, 106
  asterisk (*), 107
  B, 110
  comma (,), 110
  credit (CR), 111
  debit (DB), 111
  dollar ($), 109
  encoded-sign, 108
  I, 108
  minus (--), 109
  nine (9), 107
  period (.), 110
  plus (+), 109
  R, 108
  S, 109
  slash (/), 110
  T, 108
  V, 107
  Y, 107
  Z, 107
Picture specification, 104
PL/I compiler, 45, 372
  functions, 45
  invoking, 46
  listing file, 49
  listing options, 50
  options, 47
PLI command, 3, 46, 50, 52
  examples, 52
  in command procedure, 15
  messages
    format, 55
  obtaining information about, 5
  parameters, 46
  qualifiers, 47
  specifying libraries, 54
PLI file type, 47
PLI$LIBRARY, 55
  defining at login, 17
  multiple definitions, 55
PLI_FILE_DISPLAY structure, 263
  device attributes, 269
  ENVIRONMENT information, 265
  file attribute information, 268
PLIRTL.EXE, 65
PLISYSDEF.TLB, 55
  system service declarations, 410
Plus (+) picture character, 109
Plus sign (+)
  prefix operator, 174
POINTER attribute, 127
POINTER built-in function, 358
Pointers
  converting to offsets, 355, 408
  data type, 127
  obtaining values, 132
    POINTER built-in function, 358
  passing as actual arguments, 208
  setting value
    ADDR built-in function, 339
    ALLOCATE statement, 130
    SET option of READ, 305
  valid value, 132
  variable
    setting to null value, 355
POSINT built-in function, 358
POSINT pseudovariable, 183
Position (file), 293
  following DELETE, 315
  following READ, 306
  following REWRITE, 313
  following WRITE, 309
  record file, 301
    after keyed operation, 304
    after sequential read, 304
  stream I/O, 293
Position (string)
  stream I/O, 296
POSITION attribute, 142
Precision
  arithmetic data types, 111
  attribute, 111
  default, 112
  fixed-point binary, 98
  fixed-point decimal, 99
  floating-point data, 101
    range, 102
  pictured variables
    defined by drifting characters, 109
Prefix operators, 174
Preprocessor, 376
  diagnostic messages, 57

Preprocessor, (Cont.)
  replacement listing, 51
  statement
    %PAGE, 392
  variables, 374, 376
Preprocessor built-in functions, 398
Primary key, 327
PRINT attribute, 277
  determining if file has, 269
  specifying on OPEN, 245
PRINT command
  EDT, 20
Print files, 294
  changing page number, 183
  declaring, 277
  default line size, 277
  determining current page number, 357
  determining line number, 352
  distinguishing features, 294
  effect of PUT SKIP, 287
  information maintained, 295
  PRN-format carriage control, 295
  specifying page size, 277
Printer
  files
    handling end-of-page condition, 234
  output, 294
Printer format
  detecting, 266
PRINTER_FORMAT
  ENVIRONMENT option, 260
Priority
  of operators, 176
    table, 176
PRN-format carriage control, 295
Procedure
  declaration
    outside of procedures, 166
%PROCEDURE statement, 381, 392
PROCEDURE statement, 190
  label restriction, 223
  RETURNS option, 196
  to define a procedure, 83
Procedures, 83, 187
  declaring, 190
  entry points, 188
  external, 83, 202
  IDENT option, 191
  internal, 83
  invoking, 83
    recursively, 191
    with CALL statement, 193
    with function reference, 194
Procedures, (Cont.)
  main procedure, 83
    identifying, 191
  parameters of, 198
  preprocessor, 393
  returning from, 195
  run-time
    linking, 59
  terminating, 189
    END statement, 218
    STOP statement, 226
  using, 187
Process logical name table, 9
Process permanent files, 253
Program output
  redefining SYSPRINT, 253
  spooling to line printer, 252
Program sections
  names in compiler listing, 50
Programs
  compiling, 45, 372
  controlling execution, 210
  creating and correcting, 18
  creating, compiling, and running, 3
  documenting, 92
  elements of, 82
  executing, 67
  format of, 92
  image exit, 67
  interrupting, 69
  linking, 59
  program development, 1
    command procedures for, 13
    EDT aids, 41
  terminating
    with END statement, 218
    with RETURN statement, 196
    with STOP statement, 226
PROMPT option
  GET statement, 280
Prompts
  EDT, 23
  with GET statement, 280
Protection (file)
  determining group access, 266
  determining owner access, 266
  determining system access, 267
  determining world access, 267
Pseudovariables, 180
  rules for use, 181
Punctuation marks
  meaning to PL/I, 84
  summary of PL/I, 85
PURGE command, 12

PURGE_TYPE_AHEAD option
  GET statement, 280
PUT statement, 285
  conversion of values, 179, 402
  default file attributes, 286
  default file title, 254
  DO option, 285
  FILE option, 286
  forms, 286
  LINE option, 287
  options, 287
  order of option execution, 286
  output source, 285
  PAGE option, 286
  PUT EDIT, 288
  PUT LINE, 289
  PUT LIST, 290
  PUT PAGE, 291
  PUT SKIP, 292
  SKIP option, 287
  STRING option, 288

Qualifying reference
  for based variable, 131
QUIT command
  EDT, 20
  using, 23

R format item, 300
R picture character, 108
Range, 25
Range specifications, 25
  multiple line, 26
  single line, 25
RANK built-in function, 360
RANK preprocessor built-in function, 399
READ statement, 304
  file position following, 304
  SET option
    using, 132
  with pictured data, 369
READ_AHEAD
  ENVIRONMENT option, 260
    determining if set, 266
READ_CHECK
  ENVIRONMENT option, 260
    determining if set, 266
/READ_ONLY qualifier, 40
RECORD attribute
  determining if file has, 269
  specifying on OPEN, 245
Record I/O and unconnected arrays, 161
Record identification
  accessing a record, 318
  obtaining, 318
Record number
  maximum
    determining, 266
  relative, 321
Record size
  determining, 266
  relative files, 323
    calculating, 323
RECORD_ID option, 318
  DELETE statement, 315
  READ statement, 306
  REWRITE statement, 312
RECORD_ID_ACCESS
  ENVIRONMENT option, 260
    determining if set, 267
RECORD_ID_TO option, 318
  READ statement, 306
  REWRITE statement, 312
  WRITE statement, 309
Records
  accessing by record indentification, 318
  default length, 323
  deleting, 314
  determining size of, 266
  file
    READ with SET option, 132
  fixed-length
    calculating size of, 323
  obtaining record identification, 318
  reading, 304
  record files, 301
    deleting records, 314
    position information, 301
    reading records, 304
    statements for processing, 301
    updating records, 311
    writing records, 308
  record I/O, 301
  rewriting, 311
  variable-length
    calculating size of, 323
  writing, 308
/RECOVER qualifier, 40
RECURSIVE option
  PROCEDURE statement, 191
REFER option, 152 to 153
References
  structure-qualified, 158
  to based variable, 131
  to system services, 410
  unresolved, 60

Relational operators, 85, 174
Relative files, 321
  creating, 321
  examples, 323
  handling errors, 326
  ONKEY built-in function, 357
  organization, 321
  reading sequentially, 325
  rewinding to first occupied cell, 272
  updating, 325
  using, 323
Relative record number, 321
  maximum, 323
RENAME command, 12
Repeat count
  entering, 37
REPEAT option
  DO statement, 215
Repetition of format item, 297
REPLACE command
  EDT, 20
  using, 30
/REPLACE qualifier, 76
%REPLACE statement, 381, 396
Replication factor, 120
RESEQUENCE command
  EDT, 20
RESIGNAL built-in subroutine, 242
  using, 232
Restricted integer expression, 145
Retrieval pointers
  determining number, 266
RETRIEVAL_POINTERS
  ENVIRONMENT option, 260
%RETURN Statement, 396
%RETURN statement, 392
RETURN statement, 195
  conversion of values, 179, 402
  effect of status values, 70
  main procedure, 67
    specifying value, 70
  terminating subroutine or function, 189
Return status values
  system services, 414
Return values, 196
  specifying attributes of, 196
RETURNS
  attribute, 196
    main procedure, 70
    with ENTRY attribute, 205
  option, 196
    ENTRY statement, 192
    PROCEDURE statement, 191
Returns descriptor, 196
REVERT statement, 241
  effect on ON-unit, 230
REWIND built-in subroutine, 272
REWIND_ON_CLOSE
  ENVIRONMENT option, 260
    determining if set, 267
    specifying on CLOSE, 250
REWIND_ON_OPEN
  ENVIRONMENT option, 260
    determining if set, 267
REWRITE statement, 311
  file position following, 304
  using, 133
RMS
  condition values, 274
ROUND built-in function, 360
Routine name
  in run-time traceback, 68
Row-major order, 149
RUN command, 3, 67
  in command procedures, 15
  interrupting, 69
Run-time errors, 68
Run-time library, 65
  PL/I
    linker requirement for, 60
Run-time procedures
  linking, 59

S picture character, 109
%SBTTL statement, 381, 397
SCALARVARYING
  ENVIRONMENT option, 260, 304, 309, 312
    determining if set, 267
Scale factor, 99, 113
  binary, 112
  decimal, 112
  default, 112
  of pictured variable, 107
  specifying, 97, 99
Scope
  of entry variable, 206
  of internal variables, 124
  of names, 168
  of ON-unit, 231
  of static variables, 124
SEARCH built-in function, 361
Search order
  INCLUDE file libraries, 54
  logical name tables, 9
  object module libraries, 63
    logical name tables, 64
SEARCH preprocessor built-in function, 399

Segmented character-string keys, 331
Select range
  use with CUT function, 39
  use with structured tabs, 41
SELECT statement, 220
Selective listing control, 391 to 392
Semicolon (;)
  using as null statement, 226
SEQUENTIAL attribute
  determining if file has, 269
  specifying on OPEN, 245
Sequential files, 318
  appending records to, 318
  creating, 318
  magnetic tapes, 319
SET command
  EDT, 20
    SET TAB, 41
SET DEFAULT command, 8, 12
SET option
  ALLOCATE statement, 130
  READ statement, 305
    example, 132
Severity
  of compiler errors, 56
Shareable image file
  linker options file for, 62
SHARED_READ
  ENVIRONMENT option, 260
    determining if set, 267
SHARED_WRITE
  ENVIRONMENT option, 260
    determining if set, 267
SHOW command
  EDT, 20
SHOW LOGICAL command, 10
/SHOW qualifier, 50
SHOW TRANSLATION command, 10
  displaying logical names, 9
SIGN built-in function, 361
SIGN preprocessor built-in function, 399
SIGNAL statement, 242
SIN built-in function, 362
SIND built-in function, 362
Sine
  computing
    from degree argument, 362
    from radian argument, 362
  computing hyperbolic, 362
Single-precision floating point
  range of precision, 102
SINH built-in function, 362
SIZE built-in function, 362
SKIP format item, 300
SKIP option
  GET statement, 280, 284
    order of execution, 280
  PUT statement, 287, 292
Slash (/) picture character, 110
SOS, 18
Source files
  creating, 24
  revising, 24
  specifying in PLI command, 47
Source program format, 92
Space character, 86
SPACEBLOCK built-in subroutine, 272
SPOOL
  ENVIRONMENT option, 260
    determining if set, 267
    specifying on CLOSE, 250
Spooled devices
  obtaining device information, 268
SQRT built-in function, 363
Square root
  obtaining, 363
STARLET.OLB, 65
Startup command files, 43
  alternate, 44
Statements, 84
  alphabetic summary, 87
    preprocessor, 380
  file control, 243
  for stream I/O, 278
  format of, 84
  record I/O, 301
Static
  storage class, 123
STATIC attribute, 123
  implied by INTERNAL, 124
  with INITIAL attribute, 168
Statistics
  compiler
    including in listing, 50
Status values
  specifying in RETURN statement, 70
STOP command, 69
STOP statement, 71, 226
  effect, 67
  terminating subroutine or function, 190
Storage
  allocating
    for a based variable, 130
    for a controlled variable, 130
    within areas, 127

Storage, (Cont.)
  allocation of
    at block activation, 82
  automatic, 123
  based, 126, 129
  built-in functions, 337
  classes of, 122
  default class, 123
  defined, 141
  freeing, 131
  locating with ADDR, 133
  of arrays, 162
  of bit strings, 116
  setting null pointer, 355
  static, 123
Storage class attributes
  alphabetic summary, 89
Storage map
  in compiler listing, 50
  linker, 63
Stream
  I/O processing, 276
    file attributes for, 276
    positioning, 292
    statements for, 278
STREAM attribute, 277
  determining if file has, 269
  specifying on OPEN, 245
Stream files, 277
  GET statement, 278
  positioning, 293
  PUT statement, 285
STRING built-in function, 363
String handling
  comparing with VERIFY, 370
  COPY built-in function, 344
  functions for
    summary, 336
  HIGH built-in function, 349
  LENGTH built-in function, 351
  locating substring, 349
  LOW built-in function, 352
  replication factor, 120
  STRING built-in function, 363
  SUBSTR built-in function, 364
  TRANSLATE built-in function, 366
STRING option
  GET statement, 279
  PUT statement, 288
STRING pseudovariable, 184
Strings
  EDT search
    locating in character mode, 37
    match criteria, 26
    specifying, 25
Strings, (Cont.)
  in conversion functions, 178
Structured tabs, 41
  using, 41
Structures, 149
  concatenating with STRING, 363
  declaring, 150
    as parameters, 199
    level numbers, 149
  dimensioned
    unconnected arrays, 162
  in an array, 160
  in assignment statements, 160
  in PUT statements, 279
  initializing, 151
  level numbers, 150
  major, 150
  minor, 150
  passing as arguments, 199
    by descriptor, 208
  referring to members, 158
  specifying in assignment, 173
  structure-qualified reference, 158
  with edit-directed I/O, 298
STS$SUCCESS, 414
STS$VALUE, 414
$STSDEF text module, 414
Subdirectories
  creating, 8
  specifying, 8
  using, 8
SUBMIT command, 13
Subroutines, 188
  CALL statement, 193
  external, 202
  file-handling, 263
    alphabetic summary, 263
  libraries, 79
  terminating, 189
Subscripts, 146
  label, 223
  refering to array of structures, 161
  variable, 146
SUBSTITUTE command, 20
  using, 31
SUBSTITUTE NEXT command, 20
SUBSTITUTE NEXT command
  using, 32
SUBSTR built-in function, 364
SUBSTR preprocessor built-in
  function, 399
SUBSTR pseudovariable, 185
Substrings
  locating in string, 349
  obtaining, 364

Substrings, (Cont.)
  overlaying string variable, 185
Subtraction operator (-), 174
SUCCESS
  status return value, 71
SUPERSEDE
  ENVIRONMENT option, 260
    determining if set, 267
Symbol definitions
  for system services, 413
Symbol substitution, 15
Symbol table
  created by compiler, 46, 48
  global, 78
Symbols
  defining at login, 16
  using, 16
    in command procedures, 14
Syntax errors
  detected by compiler, 57
SYS$BINTIM system service, 417
SYS$CLREF system service, 418
SYS$COMMAND, 254
SYS$DISK, 254
SYS$ERROR, 254
SYS$INPUT, 254
  default for GET, 279
  redefining in command
    procedure, 15
SYS$LIBRARY, 55, 65
SYS$LP_LINES, 295
SYS$OUTPUT, 254
  default for PUT, 286
SYS$SETIMR system service, 418
SYS$TRNLOG system service, 415
SYS$WAITFR system service, 418
SYSIN
  default definition of, 254
  default for GET, 279
  redefining, 253
SYSPRINT
  as print file, 277
  default definition of, 254
  default for PUT, 286
  redefining, 253
System libraries
  object module, 65
  PLISYSDEF.TLB, 55
System logical name table, 9
System messages, 70
System services, 410
  arguments
    data types for input, 412
    data types for output, 413
    specifying, 411
System services, (Cont.)
  declaring, 410
  examples, 415
  parameters
    data types, 412
  symbolic definition files, 413
  testing return status, 414
  variable-length argument lists, 413
SYSTEM_PROTECTION
  ENVIRONMENT option, 260

T picture character, 108
TAB ADJUST command
  EDT, 21
Tab character, 86
TAB format item, 300
Tab stops
  in print file, 296
  with edit-directed I/O, 300
Tables
  global symbol, 78
  logical name, 9
TAN built-in function, 365
TAND built-in function, 365
Tangent
  computing
    from degree argument, 365
    from radian argument, 365
  computing hyperbolic, 365
TANH built-in function, 365
TEMPORARY
  ENVIRONMENT option, 260
    determining if set, 267
Terminals
  as print files, 294
  output to, 294
  purging type-ahead buffer, 280
  suppressing input echo, 280
Termination
  END statement, 217
  of procedures, 189
  of program execution, 67
    STOP statement, 226
Text
  cutting and pasting, 39
  defining keypad keys to insert, 42
  deleting
    in character mode, 38
    in line mode, 30
  include from other files, 390
  inserting
    in character mode, 38
    in line mode, 29
  moving
    in character mode, 39

Text
  moving, (Cont.)
    in line mode, 31
  protecting and recovering, 40
  replacing, 30
  undeleting in character mode, 38
Text buffers, 27
  specifying, 27
Text editors, 18
Text libraries, 52, 77
  creating, 77
  using, 53
Text modules
  specifying name for, 54, 77
/TEXT qualifier, 76
Time
  converting ASCII string to
    binary, 417
  specifying for ENVIRONMENT
      options, 418
  system 64-bit value, 417
TIME built-in function, 365
Time of day
  obtaining, 365
TIME preprocessor built-in
  function, 399
Timekeeping
  functions for, 337
Timer
  setting with system service, 418
TITLE option, 247, 251
  default for SYSIN, 254
  default for SYSPRINT, 254
  determining expanded value, 269
  specifying on OPEN, 245
%TITLE statement, 381, 397
TLB file type, 47, 52, 76
Traceback
  compiler information, 45
  for run-time errors, 68
    file errors, 275
    information, 68
  specifying at compile time, 48
Transfer control
  LEAVEstatement, 224
TRANSLATE built-in function,
    366
TRANSLATE preprocessor built-in
    function, 399
Translating logical names, 9
TRIM built-in function, 367
TRIM preprocessor built-in
    function, 399
TRUNC built-in function, 368
TRUNCATE
  ENVIRONMENT option, 261
    determining if set, 267
    specifying on CLOSE, 250
Truncation
  of bit strings, 404
  of character strings, 405
  of decimal value, 368
TYPE command
  EDT, 21
TYPE command
  EDT
    using, 28

Unconnected array, 161
Undeclared variables, 164
UNDEFINEDFILE condition, 238
  signaled
    error during file opening, 248
    file cannot be opened, 273, 279
    invalid file specifications, 252
    organization incompatibility, 249
UNDERFLOW condition, 239
  default PL/I action, 229
  enabling signaling of, 191
  signaled
    conversion of values, 403
UNDERFLOW option
  PROCEDURE statement, 191
Unions, 156
Unresolved references, 60
UNSPEC built-in function, 368
UNSPEC pseudovariable, 186
UNTIL option, 212
  DO REPEAT, 215
UPDATE attribute
  determining if file has, 269
  specifying on OPEN, 245
Uppercase and lowercase letters
  in identifiers, 86
User identification code
  of file's owner
    determining, 266
User-generated diagnostic
  messages, 57
  using %ERROR, 388
  using %FATAL, 388
  using %WARN, 398
User-generated diagnostoc messages
  using %INFORM, 391

V picture character, 107
VALID built-in function, 369
  using, 105

VALUE attribute
  parameter descriptor, 206
VARIABLE
  attribute
    with ENTRY attribute, 206
  option
    ENTRY attribute, 205
Variable
  preprocessor, 376
Variable-length argument lists, 413
Variable-length records
  with fixed control area, 316
Variables, 88
  addressable, 122
  area
    declaring, 127
  assigning value to, 171
  automatic, 123
  based, 126
    associating with storage, 126
    declaring, 129
    example, 135
    referring to, 131
  bit string, 118
  character string, 114
  declaring, 88, 163
  defined, 141
    criteria for declaring, 142
  entry, 206
  external, 124
  file, 244
  in begin blocks, 217
  initializing, 166
  internal, 124
  label, 224
  localizing, 83, 217
  nonaddressable, 122
  offset
    assigning values to, 129
    declaring, 128
  pictured, 103
    assigning values to, 104
    declaring, 104
    extracting values from, 105
  pointer
    declaring, 127
  pointers, 127
  static, 123
    initializing, 168
  undeclared, 164
  using as subscripts, 146
VARIANT preprocessor built-in
    function, 399 to 400
/VARIANT qualifier, 52, 400
VARYING attribute, 114
VAX-11 Run-Time Procedure
    library, 65
VAXCONDITION condition, 239
VERIFY built-in function, 370
VERIFY preprocessor built-in
    function, 399
Version numbers
  default, 255
  magnetic tapes, 319
version numbers
  rules, 7
VMSRTL.EXE, 65
Volume sets
  magnetic tapes, 320
Volume switching, 320

WARN preprocessor built-in function,
    399, 401
%WARN statement, 381, 398
Warning (severity)
  data conversion, 178, 180
  meaning to compiler, 56
    causes, 57
  suppressing messages, 52
  undeclared variables, 164
Warning message, 398
/WARNINGS qualifier, 52
WHEN clause, 220
WHILE option
  DO REPEAT, 215
  DO statement, 211
WORLD_PROTECTION
  ENVIRONMENT option, 261
WRITE command
  EDT, 21
WRITE command
  DCL, 15
  EDT
    using, 33
WRITE statement, 308
  file position following, 304
WRITE_BEHIND
  ENVIRONMENT option, 261
    determining if set, 267
WRITE_CHECK
  ENVIRONMENT option, 261
    determining if set, 267

X format item, 300
XOR operation
  defining with BOOL, 342

Y picture character, 107

Year
  obtaining current, 345

Z picture character, 107
ZERODIVIDE condition, 240

PROGRAMMING IN VAX–11 PL/I **READER'S COMMENTS**
AA-L057B-TE

Your comments and suggestions will help us in our continuous effort to improve the quality and usefulness of our handbooks.

What is your general reaction to this handbook? (format, accuracy, completeness, organization, etc.) ______________________________

____

____

____

What features are most useful? ______________________________

____

____

____

Does the publication satisfy your needs? ______________________________

____

____

What errors have you found? ______________________________

____

____

____

Additional comments ______________________________

____

____

Name ______________________________

Title ______________________________

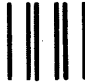
Company ______________________________ Dept. ______________

Address ______________________________

City ______________ State ______________ Zip ______________

(staple here)

— — — — Do Not Tear - Fold Here and Tape — — — — — — — — — — — — —



digital

No Postage Necessary if Mailed in the United States

BUSINESS REPLY MAIL

FIRST CLASS PERMIT NO.33 MAYNARD MASS.

POSTAGE WILL BE PAID BY ADDRESSEE

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

— — — — Do Not Tear - Fold Here — — — — — — — — — — — — — — —

Cut Along Dotted Line