DIGITAL RESEARCH™

PL/I™ Language Programmer's Guide

PL/I™
Language
Programmer's Guide

Foreword

The PL/I system is a complete software package for both applications and system programming. PL/I runs under the Digital Research single-user operating systems, CP/M® or CP/M-86®. PL/I runs in a multiuser environment, under MP/M II™, or MP/M-86™. This manual assumes you are familiar with your operating system and minimizes references to it.

PL/I is formally based on the Subset G language defined by the ANSI PL/I Standardization Committee X3J1. Subset G contains all the necessary application programming constructs of full PL/I, and discards seldom-used or redundant forms. The resulting language encourages good programming practices while simplifying the compilation task.

The *PL/I Language Programming Guide* is divided into three parts. The first part, Sections 1 through 6, presents a brief introduction to the PL/I language, with emphasis on block structure, data types, and its various executable statements. Section 5 gives guidelines for developing a readable programming style. Section 6 explains the operation of the system as a whole, and introduces you to the mechanics of compiling, linking, and executing programs.

The second part, Sections 7 through 16, contains detailed sample programs that illustrate the useful facilities of the language, including Input/Output processing, string and list processing, scientific computation, and business applications. Each section presents general concepts, and then a detailed discussion of one or more example programs to illustrate the concepts.

The third part, Section 17 through 20, presents advanced programming topics, such as the internal representation of data, conventions for interfacing assembly language programs with PL/I modules, making direct operating system calls, and writing PL/I programs that use overlays.

The best way to learn any programming language is to study working examples. To learn PL/I, you should study these example programs along with the associated text, and cross-check the material with the *PL/I Language Reference Manual* when necessary. Once you understand the operation of a particular program, you can modify the program to enhance its operation and further your experience with the language.

COPYRIGHT

DISCLAIMER

Digital Research makes no representations or warranties with respect to the contents hereof and specifically disclaims any implied warranties of merchantability or fitness for any particular purpose. Further, Digital Research reserves the right to revise this publication and to make changes from time to time in the content hereof without obligation of Digital Research to notify any person of such revision or changes.

TRADEMARKS

CP/M and CP/M-86 are registered trademarks of Digital Research. MP/M II, MP/M-86, SID and SID-86 are trademarks of Digital Research. TEX and ED are utilities of Digital Research. ADM-3A is a product of Lear-Siegler Incorporated.

The PL/I *Language Programming Guide* was prepared using the Digital Research TEX-80 Text formatter and printed in the United States of America by Commercial Press/Monterey.

First Edition: December 1982

Table of Contents

1 Introduction

1.1 What is PL/I? 1
1.2 Using This Manual 1
1.3 Notation 2

2 The PL/I Language

2.1 Structural Statements 3
2.2 Declarative Statements 3
2.3 Executable Statements 3
2.4 PL/I Blocks 5
2.5 Procedures 6
2.6 DO-groups 7

3 Declarations

3.1 Scalar Data 10
3.1.1 Arithmetic Data 10
3.1.2 String Data 12
3.1.3 Control Data 13
3.1.4 Pointer Data 16
3.1.5 File Data 16
3.2 Data Aggregates 16
3.2.1 Arrays 17
3.2.2 Structures 18

4 Executable Statements

4.1 Assignment Statements 21
4.2 Sequence Control Statements 23
4.2.1 Iteration 24
4.2.2 Procedure Invocation 26
4.2.3 Parameter Passing 26
4.2.4 Conditional Branch 29
4.2.5 Unconditional Branch 29

Table of Contents (continued)

4.3 I/O and File-handling Statements 29
4.3.1 Opening Files 30
4.3.2 File Attributes 33
4.3.3 Implied Attributes 34
4.3.4 Closing Files 35
4.3.5 File Access Methods 35
4.3.6 Data Format Items 36
4.3.7 Control Format Items 36
4.3.8 Predefined Files 37
4.4 Condition-processing Statements 37
4.4.1 The ON Statement 38
4.4.2 The REVERT Statement 38
4.4.3 The SIGNAL Statement 39
4.4.4 Condition Categories 39
4.4.5 Condition Processing Built-in Functions 41
4.5 Memory Management Statements 41
4.5.1 BASED Variables and Pointers 42
4.5.2 The ALLOCATE Statement 44
4.5.3 The FREE Statement 44
4.6 Preprocessor Statements 45
4.7 Null Statements 46

5 Programming Style

5.1 Case 49
5.2 Indentation 49
5.3 Abbreviations 50
5.4 Modular Format 50
5.5 Comments 51

6 Using the System

6.1 PL/I System Files 54
6.2 Invoking the Compiler 55

Table of Contents (continued)

6.3 Compiler Operation 57
6.4 The DEMO Program 59
6.5 Running DEMO 60
6.6 Error Messages and Codes 61
6.6.1 General Errors 62
6.6.2 Pass 1 Errors 63
6.6.3 Pass 2 Errors 64
6.6.4 Pass 3 Errors 67
6.6.5 Run-time Errors 67
6.6.6 Fatal Run-time Errors 67
6.6.7 Nonfatal Errors 68

7 Using Different Data Types

7.1 The FLTPOLY Program 71
7.2 The DECPOLY Program 74

8 STREAM and RECORD File Processing

8.1 File Copy Program 77
8.2 Name and Address File 80
8.2.1 The CREATE Program 80
8.2.2 The RETRIEVE Program 83
8.3 An Information Management System 88
8.3.1 The ENTER Program 88
8.3.2 The KEYFILE Program 91
8.3.3 The UPDATE Program 92
8.3.4 The REPORT Program 96

9 Label Constants, Variables, and Parameters

9.1 Labeled Statements 101
9.2 Program Labels 102
9.3 Computed GOTO 103
9.4 Label References 104
9.5 Example Program 105

Table of Contents (continued)

10 Condition Processing

10.1 Condition Categories 107
10.2 Condition Processing Statements 108
10.2.1 ON and REVERT 108
10.2.2 SIGNAL 110
10.3 Examples of Condition Processing 110
10.3.1 The FLTPOLY2 Program 111
10.3.2 The COPYLPT Program 112

11 Character String Processing

11.1 The OPTIMIST Program 119
11.2 A Parse Function 123
11.2.1 The GNT Procedure 125
11.2.2 The DO-Group 126

12 List Processing

12.1 Based and Pointer Variables 129
12.2 The REVERSE Program 132
12.3 A Network Analysis Program 136
12.3.1 NETWORK List Structures 139
12.3.2 Traversing the Linked Lists 141
12.3.3 Overall Program Structure 142
12.3.4 The Setup Procedure 142
12.3.5 The Connect Procedure 142
12.3.6 The Find Procedure 143
12.3.7 The Print_All Procedure 143
12.3.8 The Print_Paths Procedure 143
12.3.9 The Print_Route Procedure 143
12.3.10 The Shortest_Distance Procedure 144
12.3.11 The Free_All Procedure 144
12.3.12 NETWORK Expansion 145

Table of Contents (continued)

13 Recursive Processing

13.1 The Factorial Function 153
13.2 FIXED DECIMAL and FLOAT BINARY Evaluation 158
13.3 The Ackermann Function 161
13.4 An Arithmetic Expression Evaluator 163
13.4.1 The Exp Procedure 165
13.4.2 Condition Processing 166
13.4.3 Improvements 167

14 Separate Compilation

14.1 Data and Program Declarations 173
14.2 ENTRY Data 174
14.3 An Example of Separate Compilation 176

15 Decimal Computations

15.1 A Comparison of Decimal and Binary Operations 183
15.2 Decimal Representation 185
15.3 Addition and Subtraction 188
15.4 Multiplication 191
15.5 Division 193

16 Commercial Processing

16.1 A Simple Loan Program 197
16.2 Ordinary Annuity 201
16.2.1 Mixed Data Types 203
16.2.2 Evaluating the Present Value PV 205
16.2.3 Evaluating the Payment PMT 206
16.2.4 Evaluating the Number of Periods n 207
16.3 Loan Payment Schedule Format 208
16.3.1 Variable Declarations 215
16.3.2 Program Execution 216
16.3.3 Display Formats 217

Table of Contents (continued)

16.4 Computation of Depreciation Schedules 222
16.4.1 General Algorithms 222
16.4.2 Selecting the Schedule 231
16.4.3 Displaying the Output 232

17 Internal Data Representation

17.1 FIXED BINARY Representation 239
17.2 FLOAT BINARY Representation 240
17.3 FIXED DECIMAL Representation 243
17.4 CHARACTER Representation 244
17.5 BIT Representation 245
17.6 POINTER, ENTRY and LABEL Data 245
17.7 File Constant Representation 246
17.8 Aggregate Storage 246

18 Interface Conventions

18.1 Parameter Passing Conventions 247
18.2 Returning Values from Functions 253
18.2.1 Returning FIXED BINARY Data 254
18.2.2 Returning FLOAT BINARY Data 254
18.2.3 Returning FIXED DECIMAL Data 255
18.2.4 Returning CHARACTER Data 255
18.2.5 Returning BIT Data 255
18.2.6 Returning POINTER, ENTRY, LABEL Variables 256
18.3 Direct Operating System Function Calls 259

19 Dynamic Storage and Stack Routines

19.1 Dynamic Storage Subroutines 263
19.1.1 The TOTWDS and MAXWDS Functions 263
19.1.2 The ALLWDS Subroutine 264
19.2 The STKSIZ Function 266

Table of Contents (continued)

20 Overlays

20.1 Using Overlays in PL/I 267
20.2 Writing Overlays in PL/I 269
20.2.1 Overlay Method One 270
20.2.2 Overlay Method Two 271
20.2.3 General Overlay Constraints 273
20.3 Command Line Syntax 273

List of Tables

3-1. PL/I Data Types 9
4-1. PL/I Valid File Attributes 33
4-2. File Attributes Associated with I/O Access 34
4-3. PL/I Condition Categories and Subcodes 39
6-1. PL/I System Files 54
6-2. PL/I Compiler Options 56

List of Figures

2-1. PL/I Procedure Components 6
3-1. Arrays 17
3-2. Structure Declaration Hierarchy 19
4-1. Forms of the DO Statement 24
6-1. PL/I Program Development 53
8-1. Default Filenames in the Command Tail 78
17-1. FIXED BINARY Representation 239
17-2. Single-precision Floating-point Binary 240
17-3. Double-precision Floating-point Binary 242
17-4. Bit-string Data Representation 245
17-5. POINTER, ENTRY, and LABEL Data Storage 245
17-6. Aggregate Storage 246
18-1. PL/I Parameter Passing Mechanism 247

Table of Contents (continued)

List of Listings

2-1.	SAMPLE PL/I Program	4
3-1a.	External Procedure A	15
3-1b.	The CALL Program	15
4-1.	Parameter Passing	28
5-1.	PL/I Stylistic Conventions	51
6-1a.	Compilation of DEMO Using $N Switch	58
6-1b.	Compilation of DEMO Using $L Switch	59
6-2.	Interaction with the DEMO Program	60
6-3.	Error Traceback with the DEMO Program	61
7-1.	Polynomial Evaluation Program (FLOAT BINARY)	72
7-2.	Interaction with FLTPOLY Program	73
7-3.	Polynomial Evaluation Program (FIXED DECIMAL)	74
7-4.	Interaction with DECPOLY Program	75
8-1.	COPY (File-to-File) Program	78
8-2.	Interaction with the COPY Program	79
8-3.	CREATE Program	80
8-4.	Interaction with the CREATE Program	82
8-5.	Output from the CREATE Program	83
8-6.	RETRIEVE Program	84
8-7.	Interaction with the RETRIEVE Program	86
8-8.	The ENTER Program	89
8-9.	Interaction with the ENTER Program	90
8-10.	The KEYFILE Program	91
8-11.	Interaction with the KEYFILE Program	92
8-12.	Contents of the KEYFILE	92
8-13.	The UPDATE Program	93
8-14.	Interaction with the UPDATE Program	95
8-15.	The REPORT Program	97
8-16.	REPORT Generation to the Console	98
8-17.	REPORT Generation to a Disk File	99

Table of Contents (continued)

9-1.	Illustration of Label Variables and Constants	106
10-1.	Processing the REVERT Statement	109
10-2.	The FLTPOLY2 Program	111
10-3.	The COPYLPT Program	112
10-4.	Interaction with COPYLPT	115
10-5.	Output from COPYLPT	116
11-1.	The OPTIMIST Program	121
11-2.	Interaction with OPTIMIST	122
11-3.	The FSCAN Program	124
11-4.	Interaction with the FSCAN Program	125
12-1.	The REVERSE Program	133
12-2.	Interaction with the REVERSE Program	134
12-3.	Interaction with the NETWORK Program	137
12-4.	The NETWORK Program	145
13-1.	The IFACT Program	155
13-2.	Output from the IFACT Program	156
13-3.	The RFACT Program	156
13-4.	Output from the FACT Program	157
13-5.	The DFACT Program	158
13-6.	Output from the DFACT Program	159
13-7.	The FFACT Program	159
13-8.	Output from the FFACT Program	160
13-9.	The ACK Program	162
13-10.	Interaction with ACK Program	162
13-11.	EXPRESSION Program Using Evaluator EXPR1	164
13-12.	Interaction with EXPR1	167
13-13.	Expression Evaluator EXPR2	168
13-14.	Interaction with EXPR2	171
14-1.	Illustration of ENTRY Constants and Variables	176
14-2.	MAININVT—Matrix Inversion Main Program Module	178
14-3.	INVERT—Matrix Inversion Subroutine	180
14-4.	Interaction with the INVMAT Program	180

Table of Contents (continued)

16-1. The LOAN1 Program 199
16-2. Output from the LOAN1 Program 200
16-3. The ANNUITY Program 201
16-4. Interaction with the ANNUITY Program 203
16-5. The LOAN2 Program 209
16-6. First Interaction with LOAN2 218
16-7. Second Interaction with LOAN2 219
16-8. Third Interaction with LOAN2 220
16-9. Fourth Interaction with LOAN2 221
16-10. The DEPREC Program 223
16-11. First Interaction with DEPREC 234
16-12. Second Interaction with DEPREC 235
16-13. Third Interaction with DEPREC 236
16-14. Fourth Interaction with DEPREC 237
18-1. The DTEST Program 249
18-2. DIV2.ASM Assembly Language Program (8080) 250
18-3. DIV2. A86 Assembly Language Program (8086) 251
18-4. DTEST Output (abbreviated)..... 253
18-5. The FDTEST Program 256
18-6. FDIV2.ASM Assembly Language Program (8080) 257
18-7. FDIV2.A86 Assembly Language Program (8086) 258
18-8. D10MOD.DCL 260
18-9. FCB.DCL 261
19-1. The ALLTST Program 265
19-2. Interaction with the ALLTST Program 266
19-3. The ACKTST Program 267
19-4. Output from the ACKTST Program 268

Section 1
Introduction

1.1 What is PL/I?

PL/I is a programming language that you can use to write either applications or system-level programs. It is formally based on the ANSI General Purpose Subset (Subset G) as specified by the ANSI PL/I Standardization Committee X3J1. PL/I Subset G has the formal structure of the full language, but in some ways it is a new language, and in many ways an improved language compared to its parent.

PL/I Subset G is easy to learn and use. It is a highly portable language because its design generally ensures hardware independence. It is also more efficient and cost effective, because programs written in PL/I Subset G are easier to implement, document, and maintain.

1.2 Using This Manual

This manual is designed to help you learn PL/I by studying sample programs. If you have never programmed in a structured, high-level language such as PL/I, you should read Sections 1 through 4 first. These sections provide you with a brief introduction to the language. PL/I has features that are similar to other programming languages, but it also has its own unique constructs and syntax.

Sections 1 through 4 outline the fundamental structure and features of PL/I in an informal and conceptual framework. This summary can help you become familiar with the overall capabilities of PL/I and encourage you to use its full power.

Sections 1 through 4 are not a complete tutorial on PL/I programming in general. If you find the overview is not sufficiently detailed, you might want to read some of the books listed in Appendix E of the *PL/I Language Reference Manual.* You should also refer to the material in Sections 1 through 4 of the *PL/I Language Reference Manual.*

If you are already an experienced PL/I programmer, you might want to begin with Section 6, which describes how to compile and link programs.

1.3 Notation

The following notational conventions appear throughout this document:

- Words in capital letters are PL/I keywords or the names of PL/I programs that are described in the text.
- Words in lower-case letters or in a combination of lower-case letters and digits separated by a hyphen represent variable information for you to select. These words are described or defined more explicitly in the text.
- Example statements are given in lower-case.
- The vertical bar | indicates alternatives.
- ß represents a blank character.
- Square brackets [ ] enclose options.
- Ellipses . . . indicate that the immediately preceding item can occur once or any number of times in succession.
- Except for the special characters listed above, all other punctuation and special characters represent the actual occurrence of those characters.
- In text, the symbol CTRL represents a control character. Thus, CTRL-C means control-C. In a PL/I source program listing or any listing that shows example console interaction, the symbol ^ represents a control character.
- The acronym BIF refers to one of the PL/I built-in functions.
- Throughout this manual, program listings have brackets on the left side to illustrate and emphasize the block structure of the language.
- References to material in the *PL/I Language Reference Manual* are noted at the end of each section. The acronym LRM denotes Language Reference Manual. For example,

  References: LRM Section 3.1.1

End of Section 1

Section 2
The PL/I Language

Every PL/I program consists of one or more statements from three general categories:

- structural statements
- declarative statements
- executable statements

These categories are not mutually exclusive, but provide a convenient starting point. The following sections describe and illustrate the statements in each general category.

2.1 Structural Statements

Structural statements are the foundation of any program because they define the logical units in a program. These logical units are called blocks. When a program executes, control always flows from one logical unit to another. Logical units can contain other logical units, causing control to flow into and out of the units. You use structural statements to specify the hierarchical and logical structure in a program.

2.2 Declarative Statements

Declarative statements always occur in a logical unit defined by a structural statement, and determine the environment of a logical unit. The environment is the name and type of all the data variables available in a logical unit. Use declarative statements to specify the context of the variables you want to manipulate in a logical unit.

2.3 Executable Statements

Executable statements manipulate storage, transfer the flow of control between logical units, control the flow of data to and from I/O devices, and perform calculations. Both structural statements and declarative statements serve only to create a context for executable statements.

Listing 2-1 shows a PL/I program that illustrates statements from each category. You need not fully understand the program or the syntax of each statement at this point. The program consists of distinct blocks of statements; each block is a logical unit of control.

```
sample:
        procedure options(main);
        declare
           c character(10) varying;

        do;
          put skip list('Input: ');
          get list(c);
          c = upper(c); /* function reference */
          put skip list('Output: ',c);
        end;

        begin;
        declare
           c float binary(24);

           put skip list('Input: ');
           get list(c);
           call output(c); /* subroutine invocation */
        end;

        upper:
             procedure(c) returns(character(10) varying);
             declare
                c character(10) varying;

             return(translate(c,'ABCDEFGHIJKLMNOPQRSTUVWXYZ',
                              'abcdefghijklmnopqrstuvwxyz'));
        end upper;

        output:
             procedure(c);
             declare
                c float binary(24);

             put skip edit(c) (column(20),e(10,2));
        end output;

end sample;
```

Listing 2-1. SAMPLE PL/I Program

Every PL/I program must have a main procedure block. Although you can separately develop and compile external procedures that can be linked to and called from a main procedure, there can be only one main procedure block in a program. In Listing 2-1, the first two statements, together with the last statement, determine the outermost, or main block of the program.

2.4 PL/I Blocks

In PL/I a block can have its own local environment, and possibly an environment inherited from a containing block. A containing block is any block that contains another block. For example, in Listing 2-1 the DO-group inherits the environment of the main procedure block. However, the BEGIN block has its own local environment, even though it is contained in the main procedure block.

In PL/I there are two types of blocks:

- PROCEDURE blocks
- BEGIN blocks

You can nest either type of block. This means that you can put one block inside another, but the blocks cannot overlap. The essential difference between a PROCEDURE block and a BEGIN block is the way that PL/I executes each block in the overall program.

PL/I executes BEGIN blocks as they are encountered in the normal sequence of statements in the program. A BEGIN block ends when its corresponding END statement is encountered or when control passes outside the block. When control reaches a BEGIN block, the statements inside the block execute sequentially. Usually, when control leaves the block, it simply passes to a containing block or goes to the next sequential block.

PL/I ignores PROCEDURE blocks as they are encountered in the usual sequence of statements in the program. Control only passes to and enters a PROCEDURE block when the program invokes the procedure with a CALL statement or a function reference. A PROCEDURE block is active when the statements inside the block are executing. When the statements inside the procedure finish executing, the PROCEDURE block returns control to the point of the call.

For this reason, you can place a procedure anywhere in a program. It is good programming practice to put all procedures at the bottom of the main program. This makes debugging and maintaining a program easier.

2.5 Procedures

Every procedure consists of a procedure name, procedure header, the procedure body of zero or more statements, and an end statement. Figure 2-1 shows the components of the main procedure in the SAMPLE program.

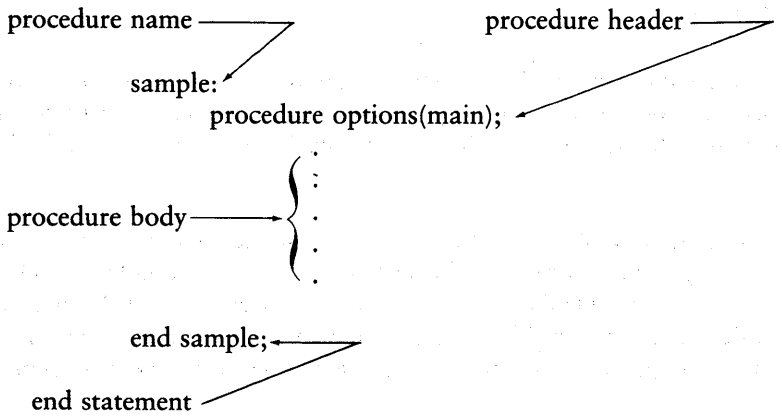


Figure 2-1. PL/I Procedure Components

If you nest procedures, they inherit the environment of containing blocks. However, any variable that you declare in a containing block can be redeclared, with local attributes, in the nested procedure.

There are two general types of procedures in PL/I:

- subroutine procedures
- function procedures

Use the CALL statement to invoke a subroutine procedure. A subroutine procedure performs a specific task, and optionally returns values to the invoking procedure.

Invoke a function procedure by making a function reference. A function reference is simply using the name of the function in a statement. PL/I evaluates the function reference and replaces it with a scalar value at the point of the reference.

Procedures are either internal or external in relation to the main procedure. An internal procedure is contained in the body of the main procedure, while external procedures are written and compiled separately from the main program. To make an external procedure known to the main procedure, you must declare the procedure name as an entry constant (see Section 3.1.3). You must also link the external procedure to the main procedure after both are compiled. All the procedures in the SAMPLE program are internal to the main procedure.

2.6 DO-groups

The DO-group is similar to the BEGIN block. There are several forms of the DO-group, and they are executable statements because they influence the flow of control. However, they are also considered structural statements because they define logical units.

Listing 2-1 illustrates the simplest form of the DO-group. It looks like a BEGIN block, but there is a crucial difference. Although a DO-group binds all the statements in its body into one logical unit, it cannot define a new environment. A DO-group cannot define new variables whose environment is limited to the body of the DO-group.

A DO-group can bind only executable statements. However, a BEGIN block can bind both declarative statements and executable statements. The environment of a DO-group is determined by the environment of the block where it occurs.

End of Section 2

Section 3
Declarations

Use declarative statements to specify the data items you want to manipulate with the executable statements in your program. PL/I has a rich variety of data types. In addition to arithmetic and string data, PL/I supports pointer, label, and entry data, which are generally not available in other languages. Table 3-1 shows the PL/I data types divided into categories and subcategories.

Table 3-1. PL/I Data Types

Category	*Subcategory*
Arithmetic	FIXED BINARY FLOAT BINARY FIXED DECIMAL
String	CHARACTER BIT
Control	Label Variable Label Constant Entry Variable Entry Constant
Pointer	POINTER
File	File Variable File Constant
Data Aggregates	Arrays Structures
Procedures	Subroutines Functions

All declarative statements specify either data constants or data variables. You must explicitly declare all data variables in a DECLARE statement, but data constants are usually declared implicitly by their occurrence in an executable statement. A PL/I variable is defined by an identifier name. The name can consist of up to thirty-one alphanumeric characters or underscores. The first character must be a letter.

Usually, declarative statements, whether explicit or implicit, result in a specific allocation of storage for the data item declared. The Compiler determines the amount of storage required for the type of data, and associates the item with this storage. BASED variables are an exception because they do not necessarily force an allocation of storage (see Section 4.5).

3.1 Scalar Data

There are two main kinds of data: scalar, or single, data items, and aggregate, or multiple, data items. Scalar data types are the fundamental data types of the language.

3.1.1 Arithmetic Data

You use arithmetic data for direct numerical calculation. PL/I provides several types of arithmetic data, so you can match the data to the application.

FIXED BINARY

You can use FIXED BINARY data to represent integers. PL/I internally represents this data type in two's complement binary form. The precision of a FIXED BINARY number is the number of bits used to represent it, independent of the sign. PL/I uses from 1 to 15 bits, so it can represent integers in the range from -32768 to +32767.

FLOAT BINARY

You can use FLOAT BINARY data to represent very small or very large numbers. FLOAT BINARY data has a binary fractional part (called the mantissa), a binary exponent, and a sign. PL/I supports both single-precision and double-precision FLOAT BINARY numbers. The precision of a FLOAT BINARY number is the number of bits in the mantissa.

Single-precision numbers can have from 1 to 24 bits, while the exponent part is always represented by 8 bits. The maximum range of single-precision FLOAT BINARY numbers in decimal is approximately 5.88 * 10**-39 to 3.40 *10**38.

Double-precision numbers can have from 24 to 53 bits, while the exponent has 11 bits. The maximum range of double-precision FLOAT BINARY numbers in decimal is approximately 9.40 * 10**-308 to 1.80 * 10**308.

FIXED DECIMAL

You can use FIXED DECIMAL data to represent numbers with a fixed decimal point. You can also use FIXED DECIMAL data to represent integers. Internally, PL/I represents FIXED DECIMAL data in binary coded decimal (BCD) digits.

FIXED DECIMAL numbers have both precision and scale. The precision is the total number of decimal digits used to represent the number. The scale is the number of decimal digits to the right of the decimal point.

In PL/I, the precision of a FIXED DECIMAL number can vary from one to fifteen, while the scale can vary from zero to fifteen. This arithmetic data type is particularly useful for commercial calculations, which require exact representations of dollars and cents and cannot accept the truncation errors of binary arithmetic.

You declare an arithmetic data variable in a declaration statement of one of the following forms, where p is the precision and q is the scale.

Statement:

DECLARE identifier FIXED BINARY[(p)];

Example:

```
declare index_counter fixed binary(7);
```

Statement:

DECLARE identifier FLOAT BINARY[(p)];

Example:

```
declare pi float binary(53);
```

Statement:

DECLARE identifier FIXED DECIMAL[(p[,q])];

Example:

```
declare base_pay fixed decimal(5,2);
```

Note: precision and scale are optional. If you omit them, PL/I supplies default values.

You should use binary arithmetic for most numerical work, because it is faster and uses the least storage. If you are doing scientific work, PL/I has a complete library of built-in mathematical functions, which includes the trigonometric and the hyperbolic functions.

3.1.2 String Data

The ability to manipulate string variables is one of the most useful features of PL/I. PL/I has a complete set of built-in functions that you can use to manipulate string data. You declare a string variable to be either a bit string or a character string in a declaration of one of the following forms:

Statement:

DECLARE identifier CHARACTER[(n)];

Example:

```
declare alphabet character(26);
```

Statement:

DECLARE identifier CHARACTER[(n)] VARYING;

Example:

```
declare state character(20) varying;
```

Statement:

DECLARE identifier BIT[(n)];

Example:

```
declare flag bit(1);
```

The VARYING attribute means that the character string can vary in length, but cannot exceed the value of n. For CHARACTER variables, the value of n can be between 0 and 254. If you want to manipulate longer strings, you can use one-dimensional arrays.

Character-string constants are implicitly declared by their occurrence in a program. Indicate a character-string constant by enclosing it in apostrophes. If you want to include an apostrophe in the string, you must precede it with an extra apostrophe. PL/I also allows you to include control characters in a character string.

The following are examples of character strings:

```
'Ada Lovelace'
'^g ^g Input Error'
'Can''t Read Previous Line'
```

Bit-string variables cannot have the VARYING attribute, and the length of a bit string cannot exceed sixteen. PL/I allows you to specify bit-string constants in several different formats. Each format corresponds to a different base, which is the number of bits used to represent the item. The formats for bit-string constants are:

- base 2 (B or B1 format)
- base 4 (B2 format)
- base 8 (B3 format)
- base 16 (B4 format)

In each of the formats, write the bit-string constant as a string of numeric digits for the desired base, enclosed in apostrophes and followed by the format type. The following are examples of the four formats:

```
'101111'B       equals      101111
'101111'B1      equals      101111
'233'B2         equals      101111
'57'B3          equals      101111
'2F'B4          equals      00101111
```

3.1.3 Control Data

There are two types of control data:

- LABEL data
- ENTRY data

LABEL data allows you to reference individual statements in your program. PL/I not only allows individual statements to have labels, it also allows you to declare label variables. This means that you can manipulate labels in your program like any other valid data items.

The value of a label variable is always a label constant, implicitly defined and declared by its occurrence as a label of a statement in the program. PL/I allows you to subscript label constants. You can also declare arrays of label variables.

You can use label variables to manipulate the flow of control between logical units of a program. It is better programming practice to do this without using GOTOs and labels.

The following program is a whimsical example of label variables.

```
chase_your_tail:
     procedure options(main);
     declare
        wherever label;

     there:
          wherever = here;
     here:
          wherever = there;
     goto wherever;

end chase_your_tail;
```

PL/I also supports a powerful data type called ENTRY data. ENTRY data allows you to reference procedures just like any valid data item. You can declare an entry variable then assign it a value. The value of an entry variable is an entry constant.

Entry constants are the labels of procedures, rather than labels of executable statements. An entry constant is implicitly declared by its appearance as a label to an internal procedure.

When you declare an entry variable, you must explicitly define the type of entry constant that the variable can assume. When you explicitly declare an entry constant, you must declare it with the same attributes as the procedure it references.

The programs shown in Listing 3-1 illustrate these concepts. Listing 3-1a shows an external procedure called a. Listing 3-1b shows the program CALL, that references a. In CALL, f is an entry variable that assumes three different constant values. To create an executable program, you compile each module separately then link them together.

```
a:
  procedure(x) returns(float); /* external procedure */
  declare x float;
  return(x/2);
end a;
```

Listing 3-1a. External Procedure A

```
call:
    procedure options(main);
    declare
       f(3) entry(float) returns(float) variable,
        a entry(float) returns(float); /* entry constant /*
    declare
       i fixed, x float;

    f(1) = a;
    f(2) = b;
    f(3) = c;

    do i = 1 to 3;
       put skip list('Type x ');
       get list(x);
       put list('f(',i,')=',f(i)(x));
    end;
    stop;

    b:
      procedure(x) returns(float); /* internal procedure */
      declare x float;
      return (2*x + 1);
    end b;


    c:
      procedure(x) returns(float); /* internal procedure */
      declare x float;
      return(sin(x));
    end c;

end call;
```

Listing 3-1b. The CALL Program

3.1.4 Pointer Data

Pointer data allows you to manipulate the storage allocated to variables. The value of a pointer variable is the address of another variable.

3.1.5 File Data

File data items describe and provide access to the data associated with an external device. File data items are either file constants or variables. You must always assign a file constant to a file variable before you access the data in the file.

Declare file data in a declaration statement in one of the following forms:

Statement:

DECLARE identifier FILE;

Example:

```
declare current_transaction file;
```

Statement:

DECLARE identifier FILE VARIABLE;

Example:

```
declare f(2) file variable;
```

The executable statements used for file access determine the file attributes. (Section 4.3 describes file-handling and I/O operations.)

3.2 Data Aggregates

A data aggregate is a combination of data types that forms a data type on a higher level. There are two kinds of aggregates in PL/I:

- arrays
- structures

3.2.1 Arrays

An array is a subscripted collection of data items of the same data type. PL/I allows arrays of arithmetic values, character strings, bit strings, label constants, entry constants, pointers, files, or structures (see Section 3.2.2).

The following are examples of array declarations:

```
declare test_scores(100);

declare A(4,5);

declare A(1:4,2:5,0:10);
```

You make direct references to individual elements of an array by using a subscripted variable reference. PL/I also allows you to make cross-sectional references, with the restriction that the reference must specify a data component whose storage is connected. For example, using the following declarations,

```
declare A(5,2) fixed binary;

declare B(5,2) fixed binary;
```

you can visualize the arrays pictured in Figure 3-1:

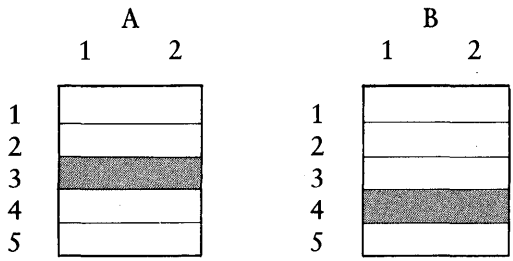


Figure 3-1. Arrays

In this example, A and B are identical in size and attributes. Therefore, an assignment such as

```
A(3) = B(4);
```

is valid because the cross-sectional reference specifies connected storage.

3.2.2 Structures

A structure is a very different type of data aggregate than an array. A structure is hierarchical, much like a tree, where the leaves of the tree, called nodes, can be various PL/I data types.

Each node of the tree, beginning with the root, has a name and a level number. The level number indicates the level of each node in relation to the root. The following example illustrates a structure declaration.

```
declare
  1 employee,
    2 name_address,
      3 name,
        4 first character(10),
        4 middle_initial character(1),
        4 last character(20),
      3 address,
        4 street character(40),
        4 city character(10),
        4 state character(2),
        4 zip_code character(5),
    2 position,
      3 department_no character(3),
      3 job_title character(20),
    2 salary fixed decimal(8,2),
    2 number_of_dependents fixed,
    2 health_plan bit(1),
    2 date_hired character(8);
```

Figure 3-2 illustrates the hierarchy of levels that corresponds to the declaration.

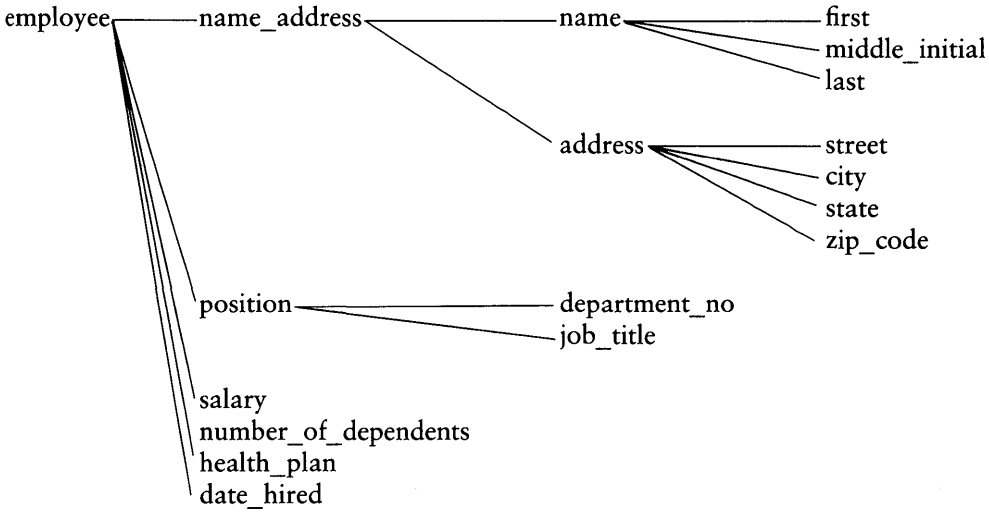


Figure 3-2. Structure Declaration Hierarchy

Nodes on each level can also determine a structure. Such a substructure is a member of the main structure. You can give the BASED attribute to the main structure with the result that all the members of the structure receive the attribute.

Structures are powerful tools because they enable you to group logically related data items that might not have the same data type. Thus, structures allow you to characterize and manipulate logical objects in your program to more closely resemble real data.

End of Section 3

Section 4
Executable Statements

The category of executable statements is divided into several subcategories based on the type of function that the statement performs. The subcategories are:

- assignment statements
- sequence control statements
- I/O and file-handling statements
- memory management statements
- condition processing statements
- preprocessor statements
- null statements

4.1 Assignment Statements

An assignment statement places the value of an expression into the storage location associated with a variable.

An expression is a combination of operators, operands, function references, and parentheses that control the order of evaluation of the expression.

In PL/I, the assignment statement has the form:

```
variable reference = expression;
```

An expression in PL/I can be fairly complicated. The simplest type of variable reference is instantiating the variable name, which means to assign the variable a specific value. A variable reference can also be a reference to a data aggregate, or to a component of the aggregate. If the variable is BASED, a pointer-qualified reference might be required (see Section 4.5.1).

PL/I also allows certain built-in functions such as UNSPEC and SUBSTR to appear as targets on the left side of assignment statements. When they appear as variables in this context, they are called pseudo-variables.

Expressions can be computational. This means that the expression involves arithmetic or string values of the various types and their respective operators. Expressions can also be noncomputational, involving comparisons of noncomputational data types such as labels, entry constants, and pointers.

PL/I allows computational expressions of different data types, and automatically performs conversion between the various types following a standard set of default rules. You should become familiar with the automatic conversion rules and the properties of the built-in conversion functions (see Section 4 LRM).

The following sequence of code illustrates some simple examples of assignment statements. These examples also illustrate some of the ways you can reference a variable in PL/I. Such references can also occur in expressions, although PL/I limits aggregate expressions to comparison for equality.

```
assign:
        procedure options(main);
        declare
          p pointer,
          i fixed binary(7),
          r bit(16),
          s bit(16) based,
          (u,v) float binary(24),
          A(5,2) character(2),
          B(5,2) character(2),
          C character(20),
          1 w,
            2 x fixed binary,
            2 y bit(16),
          1 z,
            2 x fixed binary,
            2 y bit(16);
             .
             .
             .
    u = u + v; /* simple assignment */
    A = B; /* array aggregate assignment */
    A(3) = B(4); /* cross-sectional reference */
    w = z; /* structure aggregate assignment */
    p -> s = (r = w.y); /* pointer-qualified reference */
    w.x = w.x + z.x; /* partially-qualified aggregate reference */
    unspec(w.y) = unspec(A(5,1)); /* pseudo-variable reference */
    substr(C,i+1,3) = substr(C,10,3); /* pseudo-variable reference */
    A(2*i+1) = B(4); /* variable is expression */
             .
             .
             .
end assign;
```

4.2 Sequence Control Statements

You can use sequence control statements to alter the normal sequential flow of control. In PL/I, sequence control statements perform unconditional branching, conditional branching, iteration, branch and return through procedure invocation, and a more unique construct called condition processing.

4.2.1 Iteration

PL/I provides an extensive variety of iteration control in the various forms of the DO statement. For example, you can perform iteration not only with an arithmetic control variable, but also with a pointer control variable that is moving through a linked list of pointers.

The following diagrams illustrate the basic forms of the DO statement and the flow of control that they induce. The values e1, e2, e3, and e4 represent any scalar expressions.

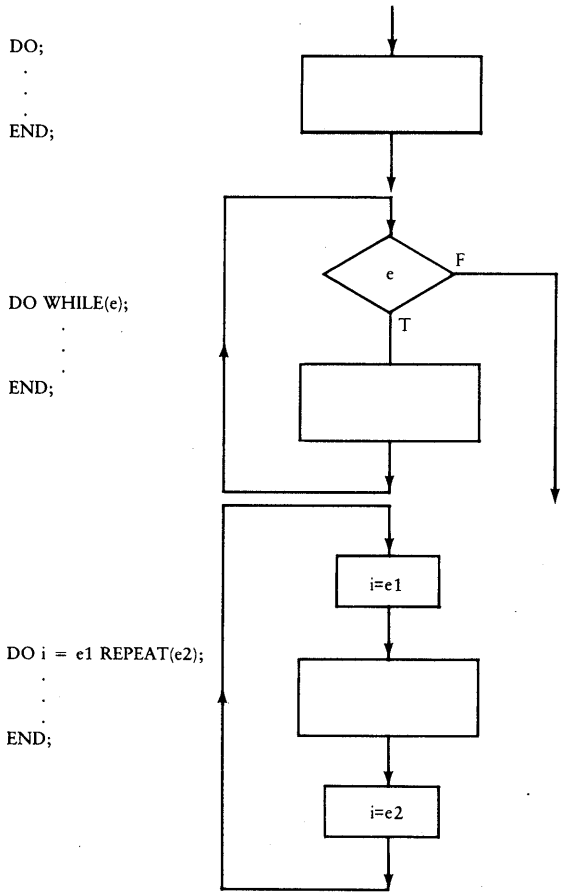


Figure 4-1. Forms of the DO Statement

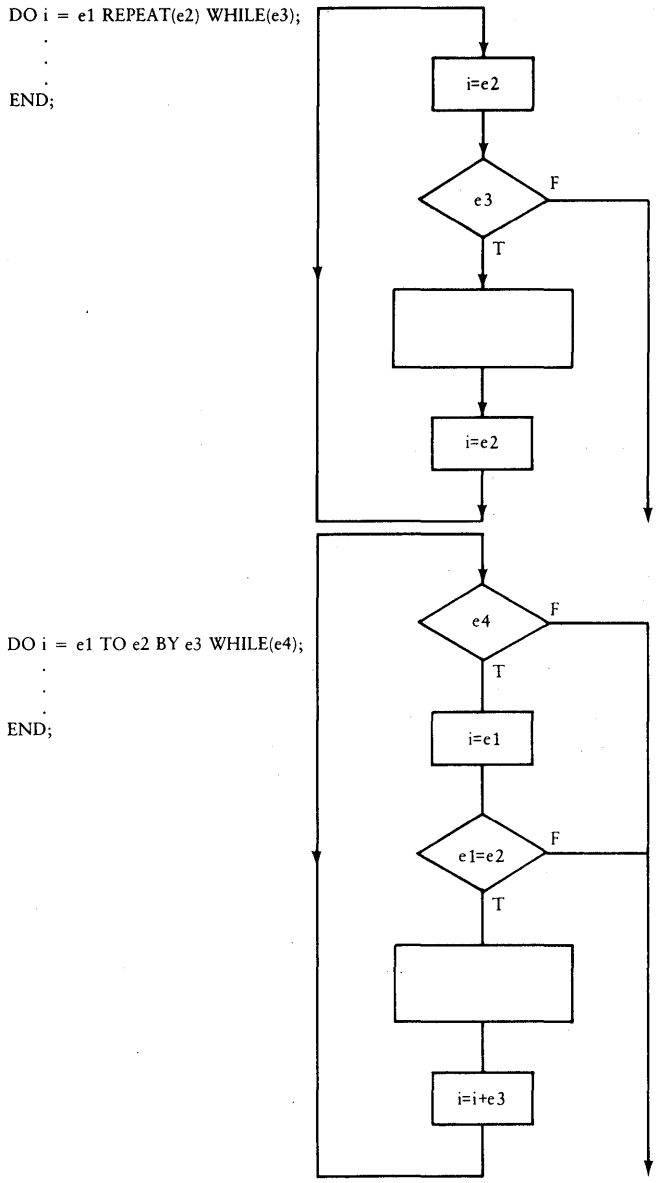


Figure 4-1. (continued)

4.2.2 Procedure Invocation

A branch and return occurs as the result of a procedure invocation. As we have seen, in PL/I there are two types of procedures: subroutine procedures and function procedures. There are two corresponding forms of invocation.

You invoke a subroutine procedure with a CALL statement, but you invoke a function procedure by using its name in an expression. You call a subroutine procedure for a specific reason, such as altering the value of variables passed to the procedure, or input and output. You always invoke a function procedure in an expression to return a scalar data item. In PL/I, both types of procedures can be recursive, which means they can call themselves.

There is an important distinction between a procedure definition and a procedure invocation. A procedure definition is a declarative statement; a procedure invocation is an executable statement. The data items you pass to the procedure when you invoke it are called the actual parameters. The actual parameters are distinguished from the formal parameters you give in the procedure definition. Thus, the actual parameters are the parameters as they are known in the invoking block, while the formal parameters are the corresponding parameters as they are known in the invoked block, the procedure.

4.2.3 Parameter Passing

In PL/I, you can pass parameters by reference or by value. You pass them by reference if the actual parameters and the formal parameters share storage. You pass them by value if the value of the formal parameter is held as a local copy of the value of the actual parameter.

Under PL/I rules, the formal parameter and actual parameter always share storage if they have identical attributes. If the actual parameter is an expression, or if its data attributes do not match those of the corresponding formal parameter, then the parameter is passed by value. PL/I passes the parameter by value if you enclose the formal parameter in parentheses in the procedure header statement of the procedure definition.

A procedure is an independent logical unit that performs a specific function. If you carefully define the specific function that the procedure performs and the parameters that it expects from the invoking environment, you can divide the design, coding, and debugging of the overall program into separate units.

If you pass a parameter by reference to conserve storage, be aware that the invoked procedure can change the value of a variable outside its local environment. If you want to assure that the procedure does not change a variable outside its local environment, then you must pass the parameters by value and use extra storage.

Parameter passing is a trade-off between the amount of storage available on your system and the level of modularity and isolation you want in your program. There are three alternatives for parameter passing, characterized as the high, middle, and low road. The skeletal program in Listing 4-1 illustrates the concepts they represent.

In the low road, you pass by reference but pay close attention to the possible side effects that can result. The advantage of this method is that it conserves storage.

In the middle road, you pass by value, enclosing the actual parameter in parentheses at the point of invocation in the CALL statement or function reference.

In the high road, you declare a duplicate variable for each formal parameter in the procedure definition. You then assign the corresponding formal parameter to its duplicate, and use the duplicate as a local copy in the procedure. Equally you can enclose the formal parameter in parentheses in the procedure header. The high road is least efficient in its use of storage.


```
main:
    procedure options(main);
    declare
       a float binary;
        .
        .
        .
    call low_sub(a); /* pass by reference */
        .
    call middle_sub((a)); /* pass by value */
        .
    call high_sub(a); /* pass by reference */
        .
        .
        .
    low_sub:
        procedure(x);
        declare
           x float binary;
        .
        .
        .
    end low_sub;

    middle_sub:
        procedure(x);
        declare
           x float binary;
        .
        .
        .
    end middle_sub;

    high_sub:
        procedure(x);
        declare
          (x,my_x) float binary;
         my_x = x; /* reassign using local variable */
          .
          .
          .
    end high_sub;

end main;
```

Listing 4-1. Parameter Passing

4.2.4 Conditional Branch

PL/I provides a conditional branch in the form of an IF statement. The conditional branch has one of the following forms,

IF condition THEN group

IF condition THEN group-1 ELSE group-2

where the condition is a scalar expression that PL/I evaluates and reduces to a single value, and the groups are either single statements, DO-groups, or BEGIN blocks.

You can nest IF statements, in which case PL/I matches each ELSE with the innermost unmatched IF-THEN pair. However, you can use NULL statements following an ELSE to force an arbitrary matching of ELSE statements with IF-THEN pairs. (See Section 4.7 NULL Statements.)

4.2.5 Unconditional Branch

PL/I provides an unconditional branch in the form of a GOTO statement. The unconditional branch has one of the following forms:

```
GOTO    label_constant;
GOTO    label_variable;
```

Because PL/I is block-structured, certain rules apply to the use of the GOTO. The target label must be in the same block containing the GOTO, or in a containing block. You cannot transfer control to an inner block.

4.3 I/O and File-handling Statements

The executable I/O statements provide PL/I with a device-independent input/output system that allows a program to transmit data between memory and external devices. To understand the I/O statements, you must first know about files and their attributes.

The collection of data elements that you transmit to or from an external device is called the data set. The corresponding internal file constant or variable is called a file.

As with other data items, you must declare files before you use them in a program. The general form of a file declaration is,

```
DECLARE file_id FILE [VARIABLE];
```

where file_id is the file identifier. If you do not include the optional VARIABLE attribute, the declaration defines a file constant. With the VARIABLE attribute, the declaration defines a file variable that can take on the value of a file constant through an assignment statement. You must assign a file constant to a file variable before you can perform any I/O operations.

4.3.1 Opening Files

PL/I requires that a file be open before performing any I/O operations on the data set. You can open a file explicitly by using the OPEN statement, or implicitly by accessing the file with the following I/O statements:

- GET EDIT
- PUT EDIT
- GET LIST
- PUT LIST
- READ
- WRITE
- READ Varying
- WRITE Varying

The general form of the OPEN statement is,

```
OPEN FILE(file_id) [file-attributes];
```

where file_id is the file identifier that appears in a FILE declaration statement, and file-attributes denotes one or more of the following:

- STREAM | RECORD
- PRINT
- INPUT | OUTPUT | UPDATE
- SEQUENTIAL | DIRECT
- KEYED
- TITLE
- ENVIRONMENT
- PAGESIZE
- LINESIZE

Multiple attributes on the same line are conflicting attributes, so you can only specify one attribute. The first attribute listed is the default attribute. All the attributes are optional; you can specify them in any order.

A STREAM file contains variable length ASCII data. You can visualize it as a stream of ASCII character data, organized into lines and pages. Each line in a STREAM file is determined by a linemark, which is a line-feed or a carriage return/line-feed pair. Each page is determined by a pagemark, which is a form-feed. Generally, you must convert the data in a STREAM file from character form to pure binary form before using it. ED automatically inserts a line-feed following each carriage return, but files that PL/I creates can contain line-feeds without preceding carriage returns. In this case, PL/I senses the end of the line when it encounters the line-feed.

A RECORD file contains binary data. PL/I accesses the data in blocks determined by a declared record size, or by the size of the data item you use to access the file. A RECORD file must also have the KEYED attribute, if you use FIXED BINARY keys to directly access the fixed-length records.

The PRINT attribute applies only to STREAM OUTPUT files. PRINT indicates that the data is for output on a line printer.

For an INPUT file, PL/I assumes that the file already exists when it executes the OPEN statement. When it executes the OPEN statement for an OUTPUT file, PL/I also creates the file. If the file already exists, PL/I first deletes it, then creates a new one.

You can read from and write to an UPDATE file. PL/I creates an UPDATE file, if it does not exist, when executing the OPEN statement. An UPDATE file cannot have the STREAM attribute.

You access SEQUENTIAL files sequentially from beginning to end, but you access DIRECT files randomly using keys. PL/I automatically gives DIRECT files the RECORD attribute. PL/I also requires you to declare all UPDATE files with the DIRECT attribute, so you can locate the individual records.

A KEYED file is simply a fixed-length record file. The key is the relative record position of the record within the file based upon the fixed record size. You must use keys to access a KEYED file. PL/I automatically gives KEYED files the RECORD attribute.

The TITLE(c) attribute defines the programmatic connection between an internal filename and an external device or a file in the operating system's file system. If you omit the TITLE attribute, PL/I assigns the default title file_id.DAT, where file_id is the file identifier specified in the OPEN statement.

If the character string c specifies a disk file, it must be in the form,

d:filename.typ;password

where the drive code d, the filetype, and the password are all optional. You must specify a filename. The filename cannot be an ambiguous wildcard reference.

You can also specify $1 or $2 for both the filename and filetype. $1 gets the first default name from the command line, $2 gets the second default name.

The ENVIRONMENT attribute defines fixed record sizes for RECORD files, internal buffer sizes, the file open mode, and the level of password protection. You can open a file in one of three modes: Locked, Read-Only, or Shared. Locked is the default mode, and means that no other user can access the file while it is open. Read-Only means that other users can access the file, but only to read it. Shared mode means that other users can also simultaneously open and access the file. You can use the built-in LOCK and UNLOCK functions to lock and unlock individual records in the file, so there are no collisions with other users.

If you assign a password to a file, you can also assign the level of protection that the password provides. The levels of protection are: Read, Write, and Delete. Read means that you must supply the password to read the file. Write means that you can read the file, but you must supply the password to write to the file. Delete means that you can read the file or write to it, but you cannot delete the file without the password.

The LINESIZE attribute applies only to STREAM files, and defines the maximum number of characters in the input or output line length. The PAGESIZE attribute applies only to STREAM OUTPUT files, and defines the number of lines per page on output.

4.3.2 File Attributes

PL/I controls all file transactions through an internal data structure called the File Parameter Block (FPB). The FPB contains information about the file, such as whether it is open or closed, the external device or file associated with the file, the current line and column, or record being accessed, and the internal buffer size. The FPB also contains a File Descriptor that describes the file's attributes. These attributes in turn describe the allowable methods of access. Table 4-1 summarizes the valid attributes that you can assign to a file, either through an explicit OPEN statement, or implicitly by an I/O access statement.

Table 4-1. PL/I Valid File Attributes

STREAM	INPUT			ENVIRONMENT	TITLE	LINESIZE
STREAM	OUTPUT			ENVIRONMENT	TITLE	LINESIZE PAGESIZE
STREAM	OUTPUT	PRINT		ENVIRONMENT	TITLE	LINESIZE PAGESIZE
RECORD	INPUT	SEQUENTIAL		ENVIRONMENT	TITLE	
RECORD	OUTPUT	SEQUENTIAL		ENVIRONMENT	TITLE	
RECORD	INPUT	SEQUENTIAL	KEYED	ENVIRONMENT	TITLE	
RECORD	OUTPUT	SEQUENTIAL	KEYED	ENVIRONMENT	TITLE	
RECORD	INPUT	DIRECT	KEYED	ENVIRONMENT	TITLE	
RECORD	OUTPUT	DIRECT	KEYED	ENVIRONMENT	TITLE	
RECORD	UPDATE	DIRECT	KEYED	ENVIRONMENT	TITLE	

4.3.3 Implied Attributes

If you do not open a file with explicit attributes, PL/I determines the attributes from the type of I/O statement you use to access the file. Table 4-2 summarizes the attributes implied by each of the I/O statements. In the following table, f is a file constant, x is scalar or aggregate data type that is not CHARACTER VARYING, and k is a FIXED BINARY key value.

Table 4-2. File Attributes Associated With I/O Access

I/O Statement	*Implied Attributes*
GET FILE(f) LIST	STREAM INPUT
PUT FILE(f) LIST	STREAM OUTPUT
GET FILE(f) EDIT	STREAM INPUT
PUT FILE(f) EDIT	STREAM OUTPUT
READ FILE(f) INTO(v)	STREAM INPUT
WRITE FILE(f) FROM(v)	STREAM OUTPUT
READ FILE(f) INTO(x)	RECORD INPUT SEQUENTIAL
READ FILE(f) INTO(x) KEYTO(k)	RECORD INPUT SEQUENTIAL KEYED ENVIRONMENT(Locked,Fixed(i))
READ FILE(f) INTO(x) KEY(k)	RECORD INPUT DIRECT KEYED ENVIRONMENT(Locked,Fixed(i))
	RECORD UPDATE DIRECT KEYED ENVIRONMENT(Locked,Fixed(i))
WRITE FILE(f) FROM(x)	RECORD OUTPUT SEQUENTIAL
WRITE FILE(f) FROM(x) KEYFROM(k)	RECORD OUTPUT DIRECT KEYED ENVIRONMENT(Locked,Fixed(i))
	RECORD UPDATE DIRECT KEYED ENVIRONMENT(Locked,Fixed(i))

4.3.4 Closing Files

The CLOSE statement disassociates the file from the external data set. The form of the CLOSE statement is,

```
CLOSE FILE(file_id);
```

where file_id is a file constant for which PL/I clears the internal buffers, records all the data on the disk, and closes the file at the operating system level. You can subsequently reopen the same file using the OPEN statement. PL/I automatically closes all open files at the end of the program or upon execution of a STOP statement.

4.3.5 File Access Methods

PL/I supports two methods of file access:

- STREAM I/O
- RECORD I/O

There are three different kinds of STREAM I/O:

- LIST-directed uses the GET LIST and PUT LIST statements, which transfer a list of data items without any format specifications.
- Line-directed uses the READ and WRITE statements, which allow you to access variable-length CHARACTER data in an unedited form. These statements might not be available in other implementations of PL/I.
- EDIT-directed uses the GET EDIT and PUT EDIT statements, which allow formatted access to character data items.

EDIT-directed I/O is similar to list-directed I/O except that it writes data into particular fields of the output line, as described by a list of format items. The data list specifies a number of values to write in fixed fields defined by the format list.

The format list can contain two kinds of format items: data format items and control format items. PL/I pairs each element of the data list with a data format item in the format list. The format item determines how PL/I interprets the data element. PL/I executes control format items as they are encountered in the format list.

You can precede any format item with a positive constant integer value, not exceeding 254, that determines the number of times to apply the format item or group of format items.

4.3.6 Data Format Items

The following examples show the various format items you can use in a GET EDIT or PUT EDIT statement.

A[(w)]

The A format reads or writes the next alphanumeric field whose width is specified by w, with truncation or blank padding on the right. If you omit w, the A format uses the size of the converted character data as a field width.

B[n][(w)]

The B format reads or writes bit-string values. n is the number of bits used to represent each digit. w is the field width that you must include on input.

E(w[,d])

The E format reads or writes a data item into a field of w characters in scientific notation, with maximum precision allowed in the field width. w must be at least 8.

F(w[,d])

The F format reads or writes fixed-point arithmetic values with a field width of w digits, and d optional digits to the right of the decimal point.

4.3.7 Control Format Items

LINE(ln)

Moves to the line ln in the data stream before writing the next data item.

COLUMN(nc)

Moves to column position nc in the data stream before reading or writing the next data item. This can flush the current line.

PAGE

Performs a page eject for PRINT files.

SKIP[(nl)]

Skips nl lines before reading or writing the next data item.

X(n)

Advances n blank characters into the data stream before reading or writing the next data item.

R(fmt)

Specifies a remote format. This means that the format is specified elsewhere in a FORMAT statement.

4.3.8 Predefined Files

PL/I has two predefined file constants called SYSIN, the console keyboard, and SYSPRINT, the console output display. These files do not need to be declared unless you make an explicit reference to them in an OPEN or I/O statement.

SYSIN has the default attributes:

```
STREAM INPUT ENVIRONMENT(Locked,Buff(128)) TITLE('$CON')
LINESIZE(80) PAGESIZE(0)
```

SYSPRINT has the default attributes:

```
STREAM PRINT ENVIRONMENT(Locked,Buff(128)) TITLE('$CON')
LINESIZE(80) PAGESIZE(0)
```

4.4 Condition-processing Statements

PL/I has several features that make it ideal for applications programming. One of these features is its capability for condition processing. In most languages, the program cannot recover from run-time error conditions, such as an invalid data conversion—control reverts to the operating system.

Because PL/I is designed as a production-programming language, it has various features that allow you to intercept run-time errors, program a response, and recover control. These features are collectively called condition processing.

PL/I provides condition processing with these executable statements:

- ON
- REVERT
- SIGNAL

4.4.1 The ON Statement

Use the ON statement to intercept and program a response to a run-time condition signaled by the system, or by the execution of a SIGNAL statement. The ON statement is an executable statement that defines the response. It has the form,

```
ON condition on-body;
```

where condition is one of the major condition categories, with or without a subcode (see Section 4.4.4). The on-body is a PL/I statement or statement group that you process when the condition occurs.

If the subcode is not present, then PL/I processes the ON statement when any of the subcode conditions occur. This is equivalent to subcode 0. The file conditions must have a file reference describing the file for which the condition is signaled.

4.4.2 The REVERT Statement

Use the REVERT statement to disable the ON condition set by the ON statement. This is important because you can have only sixteen ON conditions set without overflowing the condition code area. If overflow happens, the PL/I run-time system stops processing. The form of the REVERT statement is:

```
REVERT condition;
```

PL/I automatically reverts an ON condition set in a given block when control leaves the environment of that block.

4.4.3 The SIGNAL Statement

The SIGNAL statement allows you to activate the response for a condition. The form of the SIGNAL statement is:

```
SIGNAL condition;
```

4.4.4 Condition Categories

The condition categories describe the various conditions that the run-time system can signal or that your program can signal by executing a SIGNAL statement.

There are nine major condition categories with subcodes, some of which are system-defined, and some of which you can define yourself. Table 4-3 shows the predefined subcodes.

Table 4-3. PL/I Condition Categories and Subcodes

Type	*Meaning*
ERROR	
ERROR(0)	Any ERROR subcode
ERROR(1)	Data conversion
ERROR(2)	I/O Stack overflow
ERROR(3)	Function argument invalid
ERROR(4)	I/O Conflict
ERROR(5)	Format stack overflow
ERROR(6)	Invalid format item
ERROR(7)	Free space exhausted
ERROR(8)	Overlay error, no file
ERROR(9)	Overlay error, invalid drive
ERROR(10)	Overlay error, size
ERROR(11)	Overlay error, nesting
ERROR(12)	Overlay error, disk read error
ERROR(13)	Invalid OS call
ERROR(14)	Unsuccessful Write
ERROR(15)	File Not Open
ERROR(16)	File Not Keyed

Table 4-3. (continued)

Type	*Meaning*
FIXEDOVERFLOW	
FIXEDOVERFLOW(0)	Any FIXEDOVERFLOW subcode
OVERFLOW	
OVERFLOW(0)	Any OVERFLOW subcode
OVERFLOW(1)	Floating-point operation
OVERFLOW(2)	Float precision conversion
UNDERFLOW	
UNDERFLOW(0)	Any UNDERFLOW subcode
UNDERFLOW(1)	Floating-point operation
UNDERFLOW(2)	Float precision conversion
ZERODIVIDE	
ZERODIVIDE(0)	Any ZERODIVIDE subcode
ZERODIVIDE(1)	Decimal divide
ZERODIVIDE(2)	Floating-point divide
ZERODIVIDE(3)	Integer divide
ENDFILE	
UNDEFINEDFILE	
KEY	
ENDPAGE	

In addition to these predefined system condition subcodes, you can define certain subcodes for a specific application, test for the desired condition, and then use the SIGNAL statement to signal the condition.

4.4.5 Condition Processing Built-in Functions

PL/I provides certain built-in functions to help handle conditions when they occur. These functions are:

- ONCODE
- ONFILE
- ONKEY
- PAGENO
- LINENO

The ONCODE function returns the subcode of the most recently signaled condition, or zero if no condition has been signaled.

The ONFILE function returns the internal filename of the file involved in an I/O operation that signaled a condition.

The ONKEY function returns the value of the last key involved in an I/O operation that signaled a condition.

The PAGENO and LINENO functions return the current page number and line number for a PRINT file named as the parameter.

4.5 Memory Management Statements

Every variable in a PL/I program has a storage-class attribute. The storage class determines how and when PL/I allocates storage for a variable, and whether the variable has its own storage or shares storage with another variable.

PL/I supports three different storage classes:

- STATIC
- AUTOMATIC (the default in PL/I)
- BASED

PL/I treats AUTOMATIC storage as STATIC storage, except in procedures marked as RECURSIVE. The Compiler allocates storage for STATIC variables prior to execution, and the storage remains allocated as long as the program is running. You can use the INITIAL attribute to assign initial constant values to STATIC data items.

Note: only STATIC variables can have the INITIAL attribute, to be compatible with the ANSI Subset G PL/I standard.

Storage-class attributes are properties of scalars, arrays, major structure variables, and file variables. You cannot assign storage-class attributes to entry names, file constants, or members of data aggregates.

4.5.1 BASED Variables and Pointers

The Compiler does not allocate storage for variables with the BASED storage class. A based variable is a variable that describes storage that you must access with a pointer. The pointer is the location where the storage for the based variable begins, and the based variable itself determines how PL/I interprets the contents of the storage beginning at that location. Thus, the pointer and the based variable taken together are essentially equivalent to a nonbased variable.

You can visualize a based variable as a template that overlays the storage specified by its base. Thus, a based variable can refer to storage allocated for the based variable itself, or to storage allocated for other variables.

The format of the BASED variable declaration is,

DECLARE name BASED[(pointer-reference)];

For example,

```
declare A(5,5) character(10) based;
declare bit_vector bit(8) based(p);
```

where the pointer reference is an unsubscripted POINTER variable, or a function call, with zero arguments, that returns a POINTER value.

A pointer-qualified reference can be either implicit or explicit. When you declare a variable as BASED without a pointer reference, then each reference to the variable in the program must include an explicit pointer qualifier of the form,

pointer-exp -> variable

When you declare a variable as BASED with a pointer reference, then you can reference it without a pointer qualifier. The run-time system reevaluates the pointer reference at each occurrence of the unqualified variable using the pointer expression given in the variable declaration. The following code sequence illustrates the concept of based variables.

```
declare
    p pointer,
    a character(128),
    b(128) character(1) based(p),
    c(0:127) bit(8) based(p),
    d(64) bit(16) based(p),
    e(8,0:15) bit(8) based(p);
        .
        .
        .
p = addr(a);
        .
        .
        .
```

In this example, after pointer p is set to the address of a, each of the variables b, c, d, and e refers to the same 128 bytes of storage occupied by the variable a, although they do so in different ways. Thus, the variables b, c, d, and e overlay the variable a.

There is one important point to consider here. The overlays illustrated above depend on the method a particular processor uses to internally represent and store the data items. Such code makes a program implementation-dependent. For example, in implementations other than PL/I, the internal representation of an array could include some header bytes in addition to the bytes used to represent the data elements. In each case, you must investigate the internal representation before using based variables to overlay other data types.

4.5.2 The ALLOCATE Statement

The ALLOCATE statement explicitly allocates storage for a BASED variable. The ALLOCATE statement takes the form:

ALLOCATE based variable SET(pointer variable);

For example,

```
allocate input_buffer set(buffer_ptr);
```

The run-time system obtains sufficient memory for the based variable from the free storage area and then sets the pointer variable to the address of this memory segment.

4.5.3 The FREE Statement

The FREE statement releases the storage allocated to a BASED variable. The FREE statement takes the form:

FREE [pointer variable ->] based variable;

For example,

```
free input_buffer;
```

Note: the pointer variable reference is optional if you declared the based variable with a pointer reference.

The following code sequence illustrates the use of the ALLOCATE and FREE statements.

```
declare
   (p,q,r) pointer,
   a character based,
   b fixed based(r);
         .
         .
         .
   allocate a set(p);
   allocate b set(r);
   allocate a set(q);
         .
         .
         .
   free p -> a;
   free q -> a;
   free b;
         .
         .
         .
```

4.6 Preprocessor Statements

Preprocessor statements allow you to include other files and modify the source program at compile time.

The %INCLUDE statement copies PL/I source from another file at compile time. The %INCLUDE statement is useful for filling in declarations that are repeated throughout a program. The %INCLUDE statement takes the form:

%INCLUDE 'filespec';

For example,

```
%include 'fcb.dcl';
```


The %REPLACE statement allows you to replace identifiers by literal constants throughout the text of a PL/I program at compile time. The %REPLACE statement takes the form:

%REPLACE identifier BY literal constant;

You can put more than 1 identifier-constant pair in a single %REPLACE statement by separating the pairs with commas.

For example,

```
%replace
    true  by '1'b,
    false by '0'b;
```

4.7 Null Statements

The null statement does not perform any action. Its form is simply:

```
;
```

You can use the null statement as the target of a THEN or ELSE clause in an IF statement. In the following example,

```
if x > average then
   goto print_it;
else;
```

no action takes place when x is less than or equal to average, and the sequence of execution continues at the statement following the ELSE. As another example, consider this statement:

```
on endpage(report_file);
```

Here, no action takes place when PL/I processes the ON-unit for ENDPAGE, and the I/O statement that signaled the condition continues.

You can also use null statements to give more than one label to the same executable statement. For example,

```
A:;
B: statement-1;
   statement-2;
      .
      .
      .
```

End of Section 4

References: Section 4 LRM

Section 5
Programming Style

PL/I is a free-format language. You can write programs without regard to column positions and specific line formats. Each line can be up to 120 characters long terminated by a carriage return, and logically connected to the next line in sequence. The Compiler simply reads the source program from the first through the last line, disregarding line boundaries.

In exchange for this freedom of expression, you should adhere to some stylistic conventions, so that your programs can be easily read and understood by other programmers. A professional program not only produces the correct output, but is consistent in form and divided into logical segments that are easy to comprehend. A logically structured program is also much easier to debug. A well-constructed program is appreciated for its form and its function.

There are many stylistic conventions in use by individual programmers. The following rules illustrate one set of conventions that are used throughout the examples in this guide. Listing 5-1 illustrates the conventions presented in this section.

5.1 Case

You can write PL/I programs in either upper- or lower-case. Internally, the PL/I Compiler translates all characters, outside of string quotes, to upper-case. Using lower-case throughout programs generally improves readability.

5.2 Indentation

Use indentation throughout your program to set off various declarations and statements. To simplify indentation, the Compiler expands tabs ('I characters) to every fourth column position. Some text editors expand tabs to multiples of eight columns, so the line appears wider during the edit and display operations. The Compiler issues the TRUNC (truncate) error if the expanded line length exceeds 120 columns.

Program statements start at the outer block level in the first column position. Each successive block level, initiated by a DO-group, BEGIN, or PROCEDURE block, starts

at a new indentation level, four spaces or one tab stop. Give statements in a group the same indentation level, with procedure names and labels on a single line by themselves.

An IF statement should be directly followed by the condition and the THEN keyword, with the next statement indented on the next line. When the IF statement has an associated ELSE, start the ELSE at the same level as the IF. Indent the statement following the ELSE and place it on the next line. For declarations, place the DECLARE keyword on a single line, followed by the declared elements indented on the following line.

Avoid complicated attribute factoring because it reduces program readability. Insert blank lines when necessary to improve paragraphing and to separate logically distinct segments of the program.

5.3 Abbreviations

Many of the longer PL/I keywords have abbreviations (see the PL/I Command Summary for a complete list). Inconsistent use of abbreviations decreases readability, so use either the long or short forms, but not both. Make use of the underscore in variable names to improve readability.

5.4 Modular Format

You should divide large programs into several logical groups, or modules, where each module performs a specific primitive function. You should make these modules PL/I subroutines that are either locally or externally defined. Local subroutines become a part of the same main program or subprogram, while you can separately compile and link together external subroutines.

Place locally defined subroutines at the end of the program, so that the beginning contains only declarations and top-level statements that call the local subroutines. Neither the top-level statements nor the locally defined subroutines should exceed one or two pages in length.

While learning to program in PL/I, use a main program, with locally defined subroutines, following the form of the examples in this guide. When your application programs increase in size, however, you will find it more effective to break them into separate modules. This allows you to compile and link individual segments in pieces, thereby reducing overall development time.

5.5 Comments

Comments should become an integral part of your program. They are an essential element in making your program readable, for yourself and other programmers. Avoid introducing random comments throughout the source file, and do not nest them. Consistency is the watchword. Place your comments at the beginning of subroutines or logical statement groups. If your program is properly structured into well-defined modules, these explanatory remarks provide the information required to understand the overall purpose and operation of your program. They also simplify the task of maintaining and updating the code without introducing errors.

```
/*********************************************/
/* This program computes the largest of three */
/* FLOAT BINARY numbers x, y, and z.          */
/*********************************************/
test:
    procedure options(main);
    declare
       (a,b,c) float binary;
    put list ('Type Three Numbers: ');
    get list (a,b,c);
    put list ('The Largest Value is',max3(a,b,c));

    /* this procedure computes the largest of x, y, and z */
    max3: procedure(x,y,z) returns(float binary);
          declare
             (x,y,z,max) float binary;

          if x > y then
             if x > z then
                max = x;
             else
                max = z;
          else
             if y > z then
                max = y;
             else
                max = z;
          return(max);
    end max3;

end test;
```

Listing 5-1. PL/I Stylistic Conventions

Section 6
Using the System

Developing a PL/I program is a 3-step process:

1. Write the source file using ED or a similar text editor.
2. Compile the source file and generate the relocatable object file.
3. Link the relocatable object file with the Run-time Subroutine Library to generate an executable command file.

PL/I is a compiled language. Consequently, if you make any change to the source file, you must recompile the program. Try to divide large programs into several small modules, compile each module separately, then link them together. Small programs compile faster and use less storage for the Symbol Table.

Figure 6-1 illustrates the development process.

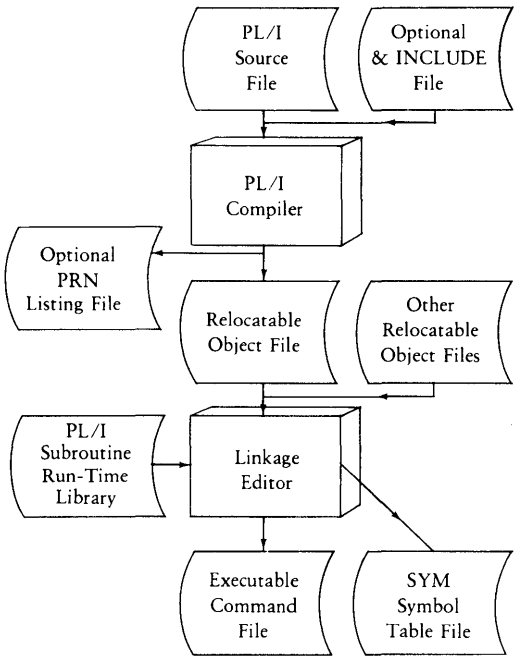


Figure 6-1. PL/I Program Development

6.1 PL/I System Files

When you receive your PL/I system you should first copy all of the files onto a back-up disk. If you are unsure how to do this, read your operating system documentation.

Note: you have certain responsibilities when making copies of Digital Research programs. Be sure you read your licensing agreement.

After you make back-up disks, load your Compiler disk and type a DIR command:

```
A>dir
```

The directory contains several types of files, as shown in Table 6-1.

Table 6-1. PL/I System Files

Type	*Definition*
CMD	Executable command file (8086 implementation), for example, DEMO.CMD
COM	Executable command file (8080 implementation), for example, DEMO.COM
DAT	Default data filetype
DCL	%INCLUDE file (data declarations)
IRL	Indexed Relocatable File, for example PLILIB.IRL
OBJ	Relocatable object code file (8086 implementation), for example, DEMO.OBJ
OVL	PL/I Compiler Overlays (8080 implementation), PLI0, PLI1, and PLI2
OVR	PL/I Compiler Overlays (8086 implementation). PLI0, PLI1, and PLI2

Table 6-1. (continued)

Type	*Definition*
PLI	PL/I source programs, for example, DEMO.PLI
PRN	Printer disk file; compiled program listing on disk
REL	Relocatable object code file (8080 implementation), for example, DEMO.REL
SYM	Symbol Table File, for example DEMO.SYM

Note: the only files that contain printable characters are the PLI source programs the PRN printer listing files and the SYM Symbol table files.

6.2 Invoking the Compiler

Invoke the PL/I Compiler using a command of the general form:

```
pli filespec [$s1...$s7]
```

where filespec designates the program to compile and can include an optional drive specification. For example,

```
d:myfile.pli
```

You need not specify the filetype because the Compiler assumes type PLI.

$s1...$s7 represent a list of parameters that you can optionally include in the command line when compiling a program. These parameters are called switches, and they enable the various Compiler options as shown in Table 6-2 on the following page.

In each case, the single-letter option follows the $ symbol in the command line. You can specify a maximum of seven options following the dollar sign. The default mode using no options compiles the program but produces no source listing and sends all error messages to the console.

Table 6-2. PL/I Compiler Options

Option	*Action Enabled*
B	Built-in subroutine trace. Shows the Run-time Subroutine Library functions that are called by your PL/I program.
D	Disk file print. Sends the listing file to disk, using the filetype PRN.
I	Interlist source and machine code. Decodes the machine language code produced by the Compiler in a pseudo-assembly language form.
K	Kill parameter and %INCLUDE listings. Disables the listing of parameters and %INCLUDE statements during the Compiler's first pass.
L	List source program. Produces a listing of the source program with line numbers and machine code locations (automatically set by the I switch).
N	Nesting level display. Enables a pass 1 trace that shows exact balance of DO, PROCEDURE, and BEGIN statements with their corresponding END statements.
O	Object code off. Disables the output of relocatable object code normally produced by the Compiler.
P	Page mode print. Inserts form-feeds every 60 lines, and sends the listing to the printer.
S	Symbol Table display. Shows the program variable names, along with their assigned, defaulted, and augmented attributes.

6.3 Compiler Operation

The PL/I Compiler reads source program files and generates a relocatable, native code object file as output. PL/I is a 3-pass Compiler, with each pass a separate overlay. Pass 1 collects declarations, and builds a Symbol Table used by subsequent passes. Pass 2 processes executable statements, augments the Symbol Table, and generates intermediate language in tree-structure form. Both passes analyze the source text using recursive descent.

Pass 3 performs the actual code generation, and includes a comprehensive code optimizer that processes the intermediate tree structures. Alternate forms of an equivalent expression are reduced to the same form, and expressions are rearranged to reduce the number of temporary variables. There is also a special-forms recognizer that detects and matches approximately three hundred tree structures of special interest. Special-forms recognition allows the Compiler to generate concise code sequences for many common statements.

Note: all the Compiler overlays (PLI0, PLI1, and PLI2) must be on the default drive.

As the Compiler proceeds through the first two passes, it displays the messages:

```
NO ERROR(S) IN PASS 1

NO ERROR(S) IN PASS 2
```

If there are errors, the Compiler lists each line containing an error with the line number to the left, a short error message, and a ? below the position in the line where the error occurs.

At the end Pass 3, the Compiler displays the message,

```
CODE SIZE = nnnn
DATA AREA = nnnn
FREE SYMS = nnnn
END  COMPILATION
```

where nnnn are hexadecimal numbers representing the amount of storage used for the code and data, as well as the amount of Transient Program Area (TPA) left for Symbol Table space.

Note: in the 8080 implementation, PL/I requires at least a 48K TPA. In the 8086 implementation, PL/I requires an 96K TPA.

If the number of error messages is excessive and you want to make corrections before proceeding, you can halt the compilation by typing a carriage return. The Compiler responds with the message:

```
STOP PL/1 (Y/N)?
```

Enter Y to halt the compilation.

If you use the N switch, the Compiler lists the program line number on the left, followed by a letter a through z that denotes the nesting level for each line. The main program level is a, and each nested BEGIN advances the level by one letter, while each nested PROCEDURE advances the level by two letters.

If you use the L switch, the Compiler lists the relative machine code address for each line as a four-digit hexadecimal number. This address is useful for determining the amount of machine code generated for each statement and the relative machine code address for each line of the program. The Compiler prints the source language statement on the line following the relative machine code value.

Listings 6-1a and 6-1b show two compilations of a program called DEMO that is on your sample program disk.

```
 1 a    demo:
 2 b        procedure options(main);
 3 b
 4 b        declare
 5 b              name character(20) varying;
 6 b
 7 b
 8 b        put skip(2) list('PLEASE ENTER YOUR FIRST NAME: ');
 9 b        get list(name);
10 b        put skip(2) list('HELLO '||name||', WELCOME TO PL/I '):
11 b
12 b    end demo;
```

Listing 6-1a. Compilation of DEMO Using $N Switch


```
 1 a 0000 ┌─demo:
 2 a 0006 │      procedure options(main);
 3 a 0006 │
 4 c 0006 │      declare
 5 c 0006 │            name character(20) varying;
 6 c 0006 │
 7 c 0006 │
 8 c 0006 │      put skip(2) list('PLEASE ENTER YOUR FIRST NAME: '
 9 c 0022 │      get list(name);
10 c 003C │      put skip(2) list('HELLO '||name||', WELCOME TO PL/I '):
11 c 006A │
12 a 006A └─end demo;
```

Listing 6-1b. Compilation of DEMO Using $L Switch

6.4 The DEMO Program

You can start learning to use the PL/I system by compiling the program called DEMO. The source file for DEMO is on your PL/I sample program disk, so you do not have to write the code. To display the source file, use the TYPE command, as follows:

```
A>type demo.pli
```

To compile the DEMO program, enter the command:

```
A>pli demo
```

Now examine your directory and find the object file that contains the relocatable machine code produced by the Compiler. The machine code produced by the Compiler is not directly executable, so you have to link the object file with the Run-time Subroutine Library (RSL) with the command:

```
A>link demo
```

Now examine your directory and find the command file and the Symbol Table file produced by the linkage editor. You can load the Symbol Table file under SID™ or SID-86™ for debugging.

6.5 Running DEMO

To run the compiled program, enter the name of the command file,

```
A>demo
```

The operating system loads the DEMO program, which begins processing and prompts you with the message,

```
PLEASE ENTER YOUR FIRST NAME:
```

Console input is free-field and incorporates the full line-editing facilities of the operating system. When you enter your name, DEMO gives an appropriate response. Listing 6-2 shows interaction with DEMO.

```
A>demo

PLEASE ENTER YOUR FIRST NAME: Larry

HELLO Larry, WELCOME TO PL/I
A>
```

Listing 6-2. Interaction with the DEMO Program

Various run-time errors can halt processing if the program does not explicitly intercept them. In this case, PL/I displays the message in the following form:

```
error-condition (code), file-option, auxiliary-message
Traceback:    aaaa bbbb cccc dddd # eeee ffff gggg hhhh
```

The error-condition is one of the standard PL/I condition categories (see Section 4.4.4). Code is an error subcode identifying the origin of the error.

PL/I prints the file option when the error involves an I/O operation, and takes the form,

```
File: internal = external
```

where internal is the internal program name that references the file involved in the error, and external is the external device or filename associated with the file. PL/I prints the auxiliary message whenever the preceding information is insufficient to identify the error.

The traceback portion lists up to eight elements of the internal stack. In the preceding general form, element aaaa corresponds to the top of the stack, while hhhh corresponds to the bottom of the stack. If the stack depth exceeds eight elements, the # character separates the four topmost elements on the left from the four lowermost elements on the right.

Listing 6-3 is an example of the diagnostic form. In this case, the console input is an end-of-file (CTRL-Z) character. Entering a CTRL-Z signals the ENDFILE condition for the SYSIN file. This is standard console input. In this example, the external device connected to the SYSIN file is the console, denoted by CON.

```
A>demo

PLEASE ENTER YOUR FIRST NAME: ^Z

END OF FILE (1), File: SYSIN=CON
Traceback: 07BE 0769 012E 4C00 # 0702 0322 8090 012E
A>
```

Listing 6-3. Error Traceback for the DEMO Program

6.6 Error Messages and Codes

PL/I prints error messages and codes during compilation and while running the compiled program. During compilation, nonfatal errors are marked with an error message following the line in error, with a ? character near the position of the line in error. The ? might follow the actual error position by a few columns. One error on a line in some cases leads to additional errors.

Fatal errors, marked with an asterisk in the following list, cause the Compiler to halt immediately. Run-time errors occur while the program is running. Although some run-time errors are fatal, most can be intercepted using ON statements. The Compiler errors are listed first.

6.6.1 General Errors

DIR FULL* The operating system's disk directory has overflowed. Erase unnecessary files and try again.

DISK FULL* All disk file space has been consumed. Erase unnecessary files and try again.

INVALID INCLUDE A %INCLUDE statement is not properly formed. The %INCLUDE statement has the general form

```
%include 'd:filename.typ';
```

where d is the (optional) drive, and filename.typ is the file specification.

LENGTH The item exceeds the maximum field width for the keyword or data item (31 characters for identifiers, 128 for strings).

NO FILE x* The file x does not exist on disk. If x is of type PLI, then check to see that your source file is on the named disk. If the type is OVR, or OVL, then ensure that all three PL/I Compiler overlays (PLI0, PLI1, PLI2) are on the default disk.

OUT OF MEMORY The memory size of the host system is too small. In the 8080 implementation, PL/I requires at least a 48K Transient Program Area (TPA) for program compilation. In the 8086 implementation, PL/I requires a 96K TPA.

READ ONLY x* The named file cannot be closed. Typically caused by disk that is set to Read-Only through hardware.

TERMINATED.* Program error count exceeds 255, or terminated at the console by user typing return during the compilation process.

TRUNC Line exceeds 120 characters in length and has been truncated.

UNEXPECTED EOF* The end of the source program was encountered before the logical end of program. Typically due to unbalanced block levels (recompile with the $n toggle for nesting trace), or unbalanced comments and strings (check balance for missing */ or apostrophe characters).

VALUE Indicates that the converted number exceeds the 16-bit capacity for FIXED BINARY constants (−32768, +32767).

6.6.2 Pass 1 Errors

BAD VAL The constant encountered in a format is invalid for this format item.

BALANCE The parentheses for the expression are not balanced.

BLOCK AT LINE x VARIABLE v EXCEEDS STORAGE The block beginning at source line x contains a variable v that caused the collective allocation of storage to exceed 65535 bytes.

BLOCK OVERFLOW The nesting level of PROCEDURE, DO, and BEGIN blocks exceeds thirty-one levels. Simplify the program structure and try again.

CONFLICT The attributes given in a declaration conflict with one another.

DUPLIC The indicated variable is declared more than once within this block.

LABEL The label for this statement is not properly formed. Only one label per statement is allowed, and subscripted label constants must have constant indexes.

LENGTH The length of the indicated symbol exceeds the maximum symbol size. Simplify the structure and retry. Can also be caused by an unbalanced string.

NESTED REP The %REPLACE statement is placed improperly in the block structure. %REPLACE statements must occur at the outer block level before the occurrence of nested inner blocks.

NO DCL: v1, v2, ..., vn The listed procedure parameters occurred in the procedure header, but were not declared within the procedure body.

NOT BIF The BUILTIN attribute is applied to an identifier that is not a PL/I built-in function.

NOT IMP The statement uses a feature that is not implemented in PL/I.

NOT VARIABLE The declared name is treated as a variable, but does not have the VARIABLE attribute.

NUMBER Numeric constant is required at this position in the format.

ON BODY Invalid statement occurs in the ON condition body. RETURN cannot be used to exit from an ON-unit. DO and IF require enclosing BEGIN...END block.

PICTURE Picture declaration or P format item is improperly formed.

RECUR PROC Recursive procedure contains invalid nested block. Only embedded DO-groups are allowed in recursive procedures.

STRUCTURE The indicated structure is improperly formed. Nesting levels cannot exceed 255.

SYMBOL LENGTH OVERFLOW* Maximum symbol size exceeded during construction of Symbol Table entry. Simplify and try again.

SYMBOL TABLE OVERFLOW* This program cannot be compiled in the current memory size. Break the module into separate compilations, or increase the size of the TPA on your system.

SYNTAX The specified statement is improperly formed. See the *PL/I Language Reference Manual* for proper statement formulation.

6.6.3 Pass 2 Errors

AGG VAL Actual parameter is an aggregate value that does not match the formal parameter. Change actual or formal parameter to match.

ARG COUNT One of the following has occurred: subscript count does not match declaration; DEFINED reference to array element; more than 15 bound pairs; bound pairs do not match; or formal and actual parameter count does not match.

BASE Invalid based variable reference. Occurs when pointer qualifier references nonbased variable, or variable is declared BASED(x), where x is not a simple pointer variable or simple pointer function call, as in BASED(P) or BASED(Q()).

BASED REQ A based variable is required in this context.

BAD TYPE Control variable in iterative DO-group is invalid. Only scalar variables are allowed.

BAD VALUE Invalid argument to built-in function.

BALANCE The parentheses for this expression are unbalanced.

BIT CON Bit substring constant is out of range. The last argument to bit SUBSTR must be a constant in the range 1 to 16.

BIT REQ A bit expression is required in this context.

CLOSURE The label following the END does not match the preceding corresponding PROCEDURE name.

COMP REQ A noncomputational expression has been used where a computational expression is required.

COMPILER ERROR A compiler error has occurred. The error might be due to previous errors.

CONFLICT Data attributes are in conflict, or attributes in OPEN statement are not compatible.

CONVERT Cannot convert the constant to the required type.

EXPRESSION OVERFLOW* The expression has overflowed the Compiler's internal structures. Simplify and try again.

ID REQ An identifier is required in this context.

INT REQ An integer (FIXED BINARY) expression is required in this context.

LABEL Improperly formed label encountered where label expected.

NO BUILTIN Referenced built-in function not implemented in PL/I.

NO DCL Indicated variable has not been declared in the scope of this reference.

NOT FILE The reference within a FILE option is not a file variable or file constant.

NOT FORMAT The format field of a GET or PUT EDIT does not reference a format.

NOT IMP The construct in this statement is not implemented in PL/I.

NOT KEY The expression within a KEYTO, KEYFROM, or KEY option is not a FIXED BINARY variable.

NOT LABEL The target of this GOTO statement is not a label value.

NOT PROC The reference following a CALL is not a procedure value.

NOT SCALAR A nonscalar value was encountered in a context requiring a scalar expression.

NOT STATIC An attempt was made to initialize automatic storage. Declare with STATIC attribute and retry.

PTR REQ A pointer variable is required in this context.

QUALIFY This reference to a structure does not properly qualify the variable name; usually due to nonunique substructure reference.

RET EXP The expression in a return statement is not compatible with the RETURNS attribute of the corresponding procedure.

RETURN An attempt to return value from procedure was made without RETURNS attribute.

SYNTAX Statement is improperly formed. See *PL/I Language Reference Manual* for proper statement formulation.

SCALE GREATER THAN 0 The resulting FIXED BINARY expression produces a nonzero scale factor. If the expression involved division, replace x/y by DIVIDE(x,y,0). This is necessary to maintain full language compatibility.

SYMBOL TABLE OVERFLOW* Free memory space exhausted during compilation. (See similar error in Pass 1.)

STR REQ A string variable is required in this context. In the case of the SUBSTR built-in function, assign the expression to a temporary variable before the substring operation takes place.

TYPES NOT= The types of a binary operation are not compatible. Check declarations and conversion rules. Might be due to aggregate data items that do not match in structure.

UNSPEC Source or target of UNSPEC operation is not an 8- or 16-bit variable.

VALUES The number of items specified in an INITIAL statement is not compatible with the variable being initialized.

VAR REQ A variable is required in this context.

6.6.4 Pass 3 Errors

AUTOMATIC STORAGE OVERFLOW The total storage defined within this program module exceeds 65535 bytes.

BAD INT FILE The intermediate file sent to Pass 3 is invalid, and is usually due to a hardware malfunction.

BLOCK OVERFLOW Nesting level has exceeded the Compiler's internal tables (maximum 32 levels).

EOF ON INT FILE Premature end-of-file encountered while reading intermediate file. Usually due to hardware failure.

EXPRESSION OVERFLOW* The Compiler's internal structure sizes have been exceeded. Simplify expression and retry.

LINE x OPERATION NOT IMPLEMENTED An invalid intermediate operation has occurred. Usually due to hardware failure or errors in a previous pass.

6.6.5 Run-time Errors

Run-time errors occur when the linked program is loaded and executed. Run-time errors are divided into two categories: fatal errors, which stop execution, and nonfatal errors, which can be intercepted with ON-units.

6.6.6 Fatal Run-time Errors

FREE REQUEST OUT OF RANGE A FREE statement specifies a storage address outside the range of the free storage area, and is usually caused by reference to an uninitialized base pointer.

FREE SPACE OVERWRITE The free storage area has been destroyed, and is usually caused by an out-of-range subscript reference or a stack overflow. If stack overflow occurs, use the STACK(n) keyword in the OPTIONS field to increase the stack size.

INSUFFICIENT MEMORY The loaded program cannot execute in the memory size allocated to the transient program. If possible, increase the size of the Transient Program Area.

INVALID I/O LIST The list of active files has been destroyed during execution, and the attempt to close all active files at the end of execution failed. Usually due to subscript values out-of-range.

6.6.7 Nonfatal Errors

The following errors are printed when no ON-unit is active, or if control returns from an ON-unit corresponding to a fatal condition (marked by an asterisk). In each case, the condition prefix is listed, followed by an optional subcode that identifies the error source, followed in some cases by an auxiliary message that further identifies the source of the error.

ERROR(1) "Conversion"* Occurs whenever conversion cannot be performed between data types, and might be signaled during arithmetic operations, assignments, and I/O processing with GET and PUT statements.

ERROR(2) "I/O Stack Overflow"* The run-time I/O stack has exceeded 16 simultaneous nested I/O operations. Simplify the source program and try again.

ERROR(3)* Transcendental function argument is out-of-range.

ERROR(4) "I/O Conflict x"* A file has been explicitly or implicitly opened with one set of attributes, and subsequently accessed with a statement requiring conflicting attributes. The value of x is one of the following:

- STREAM/RECORD
- SEQUEN/DIRECT
- INPUT/OUTPUT
- KEYED Access

The first conflict arises when ASCII files are processed using READ or WRITE, but the INTO or FROM option does not specify a varying character string.

ERROR(5) "Format Overflow"* The nesting level of embedded formats has exceeded 32. Simplify the program and try again.

ERROR(6) "Invalid Format Item"* The format processor has encountered a format item that cannot be processed. The P format is not implemented in PL/I.

ERROR(7) "Free Space Exhausted"* No more free space is available. If intercepted by an ON-unit, do not execute ALLOCATE, OPEN, or recursion without first releasing storage.

ERROR(8) "OVERLAY, NO FILE d:filename" The indicated file could not be found.

ERROR(9) "OVERLAY, DRIVE d:filename" An invalid drive code was passed as a parameter to overlay.

ERROR(10) "OVERLAY, SIZE d:filename" The indicated overlay will overwrite the PL/I stack and/or free space if loaded.

ERROR(11) "OVERLAY, NESTING d:filename" Loading the indicated overlay would exceed the maximum nesting depth.

ERROR(12) "OVERLAY, READ d:filename" Disk read error during overlay load; probably caused by premature EOF.

ERROR(13) "Invalid OS Version" Caused by any operation that generates an operating system call not supported under the current operating system.

ERROR(14) "Unsuccessful Write" Caused by any unsuccessful write operation on a file due to lack of directory space, lack of disk space, etc.

ERROR(15) "File Not Open" Caused by any attempt to lock or unlock a record in a file that is not open.

ERROR(16) "File Not Keyed" Caused by any attempt to lock or unlock a record in a file that does not have the KEYED attribute.

FIXEDOVERFLOW A decimal add or multiply produced a value exceeding 15 decimal digits of precision, or an attempt was made to store to a variable with insufficient precision.

OVERFLOW(1) A floating-point operation produced a value too large to be represented in floating-point format.

OVERFLOW(2) A double-precision floating-point value has been assigned to a single-precision value with insufficient precision.

UNDERFLOW(1) A floating-point operation produced a value too small to be represented in floating-point format.

UNDERFLOW(2) A double-precision floating-point value has been assigned to a single-precision value with insufficient precision.

ZERODIVIDE(1) A decimal divide or modulus operation was attempted with a divisor of zero.

ZERODIVIDE(2) A floating-point divide or modulus operation was attempted with a divisor of zero.

ZERODIVIDE(3) An integer divide or modulus operation was attempted with a divisor of zero.

ENDFILE An attempt was made to read past the end of the listed file, or the disk full condition occurred during output.

UNDEFINEDFILE The named file cannot be found on the disk if input, or cannot be created if output. Also occurs when an input device is opened for output, or an output device is opened for input.

KEY(1) Invalid key detected in output operation.

KEY(2) Invalid key encountered during input operation.

ENDPAGE An end-of-page condition was detected. This condition does not cause termination if no ON-unit is active.

End of Section 6

Section 7
Using Different Data Types

PL/I programs allow you to use different data types to suit different applications. In programs throughout the manual, you should note how and why each type of data is used in a particular situation.

7.1 The FLTPOLY Program

Listing 7-1 shows a program for evaluating a polynomial expression. The program begins by reading three values, x, y, and z, from the console, and then uses the values to evaluate the polynomial expression:

$$p(x,y,z) = x^2 + 2y + z$$

The main part of the program is bounded by a single DO-group. On each successive iteration, the program reads the values of x, y, and z from the standard SYSIN, console, file. The program then writes the value produced by p(x,y,z) to the SYSPRINT file, again, the console file. Finally, if all the input values are zero, the program executes the STOP statement and ends the indefinite loop.

The program uses the %REPLACE statement on line 8 to define the literal value of true as the bit-string constant, '1'b. The Compiler substitutes this value whenever it encounters the name true. Thus, the Compiler interprets the DO-group beginning on line 13 as,

```
do while('1'b);
    .
    .
    .
end;
```

which loops until it executes the contained STOP statement. Using %REPLACE statements to define constants can improve the readability of your programs.

```
 1 a   /*************************************************/
 2 a   /* This program evaluates a polynomial expression */
 3 a   /* using FLOAT BINARY data.                       */
 4 a   /*************************************************/
 5 a   fltpoly:
 6 b       procedure options(main);
 7 b
 8 b       %replace
 9 b           true  by '1'b;
10 b       declare
11 b           (x,y,z) float binary(24);
12 b
13 c       do while(true);
14 c          put skip(2) list('Type x,y,z: ');
15 c          get list(x,y,z);
16 c
17 c          if x=0 & y=0 & z=0 then
18 c             stop;
19 c
20 c          put skip list('        2');
21 c          put skip list('       x  + 2y + z =',P(x,y,z));
22 c       end;
23 b
24 b       P:
25 c          procedure (x,y,z) returns (float binary(24));
26 c          declare
27 c                (x,y,z) float binary;
28 c          return (x * x + 2 * y + z);
29 c       end P;
30 b
31 b   end fltpoly;
```

Listing 7-1. Polynomial Evaluation Program (FLOAT BINARY)

Listing 7-2 shows the console interaction with the FLTPOLY program. The initial values for x, y, and z are: 1.4, 2.3, and 5.67, but on the next loop, the input takes the form:

```
,4,5,,
```

This form changes the value of y only. Thus, on this loop, the values of x, y, and z are 1.4, 4.5, and 5.67. The third input line changes y and z, while the fourth line changes x only.

```
A>fltpoly

Type x,y,z:  1.4, 2.3, 5.67

       2
      x  + 2y + z =  1.223000E+01

Type x,y,z:  , 4.5,,

       2
      x  + 2y + z =  1.663000E+01

Type x,y,z:  , ,6e-3, 7

       2
      x  + 2y + z =  0.896119E+01

Type x,y,z:  2.3,,,

       2
      x  + 2y + z =  1.229119E+01

Type x,y,x:  0,0,0

A>
```

Listing 7-2. Interaction with FLTPOLY Program

7.2 The DECPOLY Program

Listing 7-3 shows the DECPOLY program, which is essentially the same program as Listing 7-1. The difference between the two programs is that FLTPOLY uses FLOAT BINARY data items, while DECPOLY uses FIXED DECIMAL items. FLOAT BINARY computations execute significantly faster than their FIXED DECIMAL equivalents, but single-precision FLOAT BINARY computations involve truncation errors, and produce an answer with only about 7 decimal places of accuracy.

```
 1 a   /*************************************************/
 2 a   /* This program evaluates a polynomial expression */
 3 a   /* using FIXED DECIMAL data.                      */
 4 a   /*************************************************/
 5 a   decpoly:
 6 b       procedure options(main);
 7 b
 8 b       %replace
 9 b           true  by '1'b;
10 b       declare
11 b           (x,y,z) fixed decimal(15,4);
12 b
13 c       do while(true);
14 c          put skip(2) list('Type x,y,z: ');
15 c          get list(x,y,z);
16 c
17 c          if x=0 & y=0 & z=0 then
18 c             stop;
19 c
20 c          put skip list('        2');
21 c          put skip list('       x  + 2y + z =',P(x,y,z));
22 c       end;
23 b
24 b       P:
25 c         procedure (x,y,z) returns (fixed decimal(15,4));
26 c         declare
27 c               (x,y,z) fixed decimal(15,4);
28 c         return (x * x + 2 * y + z);
29 c       end P;
30 b
31 b   end decpoly;
```

Listing 7-3. Polynomial Evaluation Program (FIXED DECIMAL)

Listing 7-4 shows the console interaction with the DECPOLY program. The initial input values for x, y, and z are: 1.4, 2.3, and 5.67. These are the same values used for the FLTPOLY program, but notice the difference in the output. The second loop changes the values of y and z, and the third loop changes all three values.

```
A>decpoly

Type x,y,z:  1.4, 2.3, 5.67

       2
      x  + 2y + z =            12.2300

Type x,y,z:  , .0006, 7

       2
      x  + 2y + z =             8.9612

Type x,y,z:  723.445, 80.54, 0

       2
      x  + 2y + z =        523533.7480

Type x,y,z:  0,0,,

A>
```

Listing 7-4. Interaction with DECPOLY Program

Experiment with these two programs by comparing the results when you enter the same values in each one. Then read Section 17, which describes the internal data representation for all PL/I data types. Understanding how PL/I internally treats the different data types helps you choose the right type of data to suit the application.

End of Section 7

References: Section 3.1 LRM

Section 8
STREAM and RECORD File Processing

The example programs in this section illustrate STREAM and RECORD file processing using the various I/O statements.

8.1 File Copy Program

Listing 8-1 shows a general purpose, file-to-file copy program. The program first defines and opens two file constants called input_file and output_file. It then begins executing a continuous loop that reads data from input_file and copies it to output_file.

Both OPEN statements define STREAM files with internal buffers of 8192 characters each. In the first OPEN statement, PL/I supplies the default attribute INPUT, while the second OPEN statement explicitly specifies an OUTPUT file. Otherwise, it would also default to an INPUT file.

This program shows the special use of READ and WRITE statements to process STREAM files. The READ statement on line 19 reads a STREAM file into buff, a character string of varying length. It reads each line of input up to and including the next carriage return line-feed into buff, and sets the length of buff to the amount of data read, including the carriage return line-feed character. The WRITE statement performs the opposite action. It sends the data to a STREAM file from buff. The output file receives all characters from the first position through the length of buff.

The program terminates by reading through the input file until it reaches the end-of-file (CTRL-Z) character. PL/I automatically closes all open files, and writes the internal buffers onto the disk, thus preserving the newly created output file.

```
 1 a   /************************************************/
 2 a   /* This program copies one file to another using */
 3 a   /* buffered I/O.                                 */
 4 a   /************************************************/
 5 a   copy:
 6 b       procedure options(main);
 7 b       declare
 8 b          (input_file,output_file) file;
 9 b
10 b       open file (input_file) stream
11 b            environment(b(8192)) title('$1,$1');
12 b
13 b       open file (output_file) stream output
14 b            environment(b(8192)) title('$2,$2');
15 b       declare
16 b          buff character(254) varying;
17 b
18 c       do while('1'b);
19 c          read  file (input_file)  into (buff);
20 c          write file (output_file) from (buff);
21 c       end;
22 b   end copy;
```

Listing 8-1. COPY (File-to-File) Program

Listing 8-2 shows a sample execution of the copy program using the following command line:

```
A>copy copy.pli $con
```

In this case, the input file is COPY.PLI, the original source file, while the output file is the system console. Thus, the program simply lists COPY.PLI at the terminal.

The TITLE options connect the internal filenames to external devices and files. The command line has two parts: the command itself, and the command tail, which can contain two filenames.

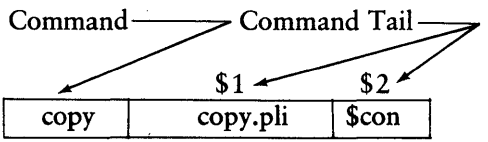


Figure 8-1. Default Filenames in the Command Tail

The OPEN statement on line 10 takes the first default name, including the drive in the command tail (denoted by $1.$1), and assigns it to the internal file constant called input_file. Similarly, the second OPEN statement on line 13 takes the second default name including the drive in the command tail (denoted by $2.$2), and assigns it to the internal file constant called output_file.

For example, the command,

```
A>copy a:x.dat c:u.new
```

copies the file X.DAT from drive a to the new file U.NEW on drive c. The input file must exist, but PL/I erases the output file if it exists, and recreates it.

```
A>copy copy.pli $con
 1 a   /*************************************************/
 2 a   /* This program copies one file to another using */
 3 a   /* buffered I/O.                                 */
 4 a   /*************************************************/
 5 a   copy:
 6 b       procedure options(main);
 7 b       declare
 8 b          (input_file,output_file) file;
 9 b
10 b       open file (input_file) stream
11 b            environment(b(8192)) title('$1.$1');
12 b
13 b       open file (output_file) stream output
14 b            environment(b(8192)) title('$2.$2');
15 b       declare
16 b          buff character(254) varying;
17 b
18 c       do while('1'b);
19 c          read  file (input_file)  into (buff);
20 c          write file (output_file) from (buff);
21 c       end;
22 b   end copy;

END OF FILE (3), File: INPUT=COPY.PLI
Traceback : 044B 03AF 0155
A>
```

Listing 8-2. Interaction with the COPY Program

8.2 Name and Address File

The two programs in Listings 8-3 and 8-6 manage a simple name and address file. The CREATE program produces a STREAM file containing individual names and addresses that are subsequently accessed by the RETRIEVE program.

8.2.1 The CREATE Program

The CREATE program in Listing 8-3 contains a data structure that defines the name, address, city, state, zip code, and phone number format. This data structure is not in the source file CREATE.PLI. It is contained in a separate file named RECORD.DCL, and CREATE uses an %INCLUDE statement to read and merge this file with the source file. Both files are on your sample program disk. The + symbols to the right of the source line number in the listing indicate that the code comes from an %INCLUDE file. The actual line in the source program appears as follows:

```
create:
       procedure options(main);

%include `record.dcl';
```

The file specified in the %INCLUDE statement can be any valid filename. The Compiler simply copies the file at the point of the %INCLUDE statement, and then continues.

The OPEN statement, line 29, does not specify the PRINT attribute. This means the output file is in a form suitable for later input using a GET LIST statement.

```
 1 a   /*****************************************************/
 2 a   /* This program creates a name and address file. The */
 3 a   /* data structure for each record is in the %INCLUDE */
 4 a   /* file RECORD.DCL.                                  */
 5 a   /*****************************************************/
 6 a ┌─create:
 7 b │       procedure options(main);
 8 b │
```

Listing 8-3. CREATE Program

```
 9+b       declare
10+b           1 record,
11+b             2 name  character(30) varying,
12+b             2 addr  character(30) varying,
13+b             2 city  character(20) varying,
14+b             2 state character(10) varying,
15+b             2 zip   fixed decimal(6),
16+b             2 phone character(12) varying;
17 b    %replace
18 b      true by '1'b,
19 b      false by '0'b;
20 b
21 b       declare
22 b          output file,
23 b          filename character(14) varying,
24 b          eofile bit(1) static initial(false);
25 b
26 b       put list ('Name and Address Creation Program, File Name: ');
27 b       get list (filename);
28 b
29 b       open file(output) stream output title(filename);
30 b
31 c       do while (^eofile);
32 c          put skip(3) list('Name: ');
33 c          get list(name);
34 c          eofile = (name = 'EOF');
35 c          if ^eofile then
36 d             do;
37 d             /* write prompt strings to console */
38 d                  put list('Address: ');
39 d                  get list(addr);
40 d                  put list('City, State, Zip: ');
41 d                  get list(city, state, zip);
42 d                  put list('Phone: ');
43 d                  get list(phone);
44 d
45 d             /* data in memory, write to output file */
46 d                  put file(output)
47 d                      list(name,addr,city,state,zip,phone);
48 d                  put file(output) skip;
49 d             end;
50 c       end;
51 b       put file(output) skip list('EOF');
52 b       put file(output) skip;
53 b
54 b    end create;
```

Listing 8-3. (continued)

Listing 8-4 shows the console interaction with the CREATE program. You specify the output file as names.dat in the first input line. The GET LIST statement, line 33, accepts input delimited by blanks and commas, unless the delimiters are included in single apostrophes. Thus, CREATE takes the input line,

```
'John Robinson
```

as a single string value with PL/I automatically inserting the implied closing apostrophe at the end of the line. The last entry includes the three input values,

```
Unknown, 'Can''t Find', 99999
```

that CREATE assigns to the variables city, street, and state. Because the first value does not begin with an apostrophe, the I/O system scans the data item until the next blank, tab, comma, or end-of-line occurs. The second data item begins with an apostrophe, and this causes the I/O system to consume all input through the trailing balanced apostrophe, and reduce all embedded double apostrophes to a single apostrophe. The last value, 99999, is assigned to a decimal number, and must contain only numeric data.

You can use the command,

```
A>type names.dat
```

to display the STREAM file that the program creates. Listing 8-5 shows the output resulting from each input entry.

```
A>create
Name and Address Creation Program, File Name:  names.dat


Name:  'Arthur Jackson'
Address:  '100 W. 3rd St.'
City, State, Zip:  'Fresno', 'Ca.', 93706
Phone:  '529-1277'


Name:  'Donna Harris'
Address:  '2999 Serra Rd.'
City, State, Zip:  'Chico', 'Ca.', 95926
Phone:  '635-3570'
```

Listing 8-4. Interaction with the CREATE Program

```
Name:  'John Robinson
Address:  '805 Franklin St,'
City, State, Zip:  'Monterey', 'Ca,', 93940
Phone:  '649-1000'


Name:  'Virginia Wilson'
Address:  '?'
City, State, Zip:  UnKnown, 'Can''t Find', 99999
Phone:  '?'


Name:  'EOF'

A>
```

Listing 8-4. (continued)

```
A>type names.dat
'Arthur Jackson' '100 W. 3rd St,' 'Fresno' 'Ca,' 93706 '529-1277'
'Donna Harris' '2999 Serra Rd,' 'Chico' 'Ca,' 95926 '635-3570'
'John Robinson' '805 Franklin St,' 'Monterey' 'Ca,' 93940 '649-1000'
'Virginia Wilson' '?' 'UnKnown' 'Can''t Find' 99999 '?'

'EOF'
A>
```

Listing 8-5. Output from the CREATE Program

8.2.2 The RETRIEVE Program

The RETRIEVE program shown in Listing 8-6 reads the file created by CREATE, and displays the name and address data upon user request. The Compiler includes the same RECORD.DCL file used in the CREATE program, shown in Listing 8-3.

The main DO-group in the RETRIEVE program, between lines 30 and 59, reads two string values corresponding to the lowest and highest names to print on each iteration. The embedded DO-group between lines 41 and 57 reads the entire input file and lists only those names between the lower and upper bounds.

The RETRIEVE program, similar to the CREATE program, reads the name of the source file from the console. However, RETRIEVE opens and closes this source file each time it receives a retrieval request from the console.

The OPEN statement on line 38 sets the internal buffer size of the input file to 1024 bytes. After processing the file, RETRIEVE executes the CLOSE statement on line 58 and flushes all internal buffers. Thus, RETRIEVE sets the input file back to the beginning on each retrieval request.

```
 1 a   /*************************************************/
 2 a   /* This program reads a name and address data file */
 3 a   /* and displays the information on request.        */
 4 a   /*************************************************/
 5 a   retrieve:
 6 b       procedure options(main);
 7 b
 8+b       declare
 9+b           1 record,
10+b             2 name  character(30) varying,
11+b             2 addr  character(30) varying,
12+b             2 city  character(20) varying,
13+b             2 state character(10) varying,
14+b             2 zip   fixed decimal(6),
15+b             2 phone character(12) varying;
16 b   %replace
17 b       true  by '1'b,
18 b       false by '0'b;
19 b
20 b       declare
21 b           (sysprint, input) file,
22 b           filename character(14) varying,
23 b           (lower, upper) character(30) varying,
24 b           eofile bit(1);
25 b
26 b       open file(sysprint) print title('$con');
27 b       put list('Name and Address Retrieval, File Name: ');
28 b       get list(filename);
29 b
30 c       do while(true);
31 c          lower = 'AAAAAAAAAAAAAAAAAAAAAAAAAAAAAA';
32 c          upper = 'zzzzzzzzzzzzzzzzzzzzzzzzzzzzzz';
33 c          put skip(2) list('Type Lower, Upper Bounds: ');
34 c          get list(lower,upper);
35 c          if lower = 'EOF' then
36 c             stop;
37 c
```

Listing 8-6. RETRIEVE Program

```
38 c        open file(input) stream input environment(b(1024))
39 c                 title(filename);
40 c        eofile = false;
41 d        do while (^eofile);
42 d           get file(input) list(name);
43 d           eofile = (name = 'EOF');
44 d           if ^eofile then
45 e              do;
46 e                 get file(input)
47 e                 list(addr,city,state,zip,phone);
48 e                 if name >= lower & name <= upper then
49 f                    do;
50 f                       put page skip(3)list(name);
51 f                       put skip list(addr);
52 f                       put skip list(city,state);
53 f                       put skip list(zip);
54 f                       put skip list(phone);
55 f                    end;
56 e              end;
57 d        end;
58 c        close file(input);
59 c     end;
60 b
61 b  end retrieve;
```

Listing 8-6. (continued)

Listing 8-7 shows user interaction with the RETRIEVE program. Again, the input file is names.dat, and exists on the disk in the form produced by CREATE. The input values,

```
B,E
```

set lower to B and upper to E and cause RETRIEVE to list only Donna Harris. The second console input line sets lower to B and upper to K. This causes RETRIEVE to list Donna Harris and John Robinson. The comma in the next input value sets the lower bound at AAA...A and the upper bound as K. Thus RETRIEVE lists Arthur Jackson, Donna Harris, and John Robinson. The last entry consists only of a comma pair, leaving the lower bound as the sequence AAA...A and the upper bound at zzz...z. These two bounds include the entire alphabetic range, so that RETRIEVE displays the entire list of names and addresses. Finally, entering EOF ends the program.

Line 26 of Listing 8-6 opens the SYSPRINT file with the PRINT attribute and title of $CON. It is good programming practice to open all files with explicit attributes. In this case the statement is redundant because when PL/I executes the PUT LIST statement on line 27, it supplies the same attributes to the file by default.


```
A>retrieve
Name and Address Retrieval, File Name:  names.dat

Type Lower, Upper Bounds:  B,E

Donna Harris
2999 Serra Rd.
Chico Ca.
    95926
635-3570

Type Lower, Upper Bounds:  B,K

Donna Harris
2999 Serra Rd.
Chico Ca.
    95926
635-3570

John Robinson
805 Franklin St.
Monterey Ca.
    93940
649-1000

Type Lower, Upper Bounds:  ,K

Arthur Jackson
100 W. 3rd St.
Fresno Ca.
    93706
529-1277

Donna Harris
2999 Serra Rd.
Chico Ca.
    95926
635-3570
```

Listing 8-7. Interaction with the RETRIEVE Program

```
John Robinson
805 Franklin St.
Monterey Ca.
    93940
649-1000

Type Lower, Upper Bounds:  ,,


Arthur Jackson
100 W. 3rd St.
Fresno Ca.
    93706
529-1277



Donna Harris
2999 Serra Rd.
Chico Ca.
    95926
635-3570



John Robinson
805 Franklin St.
Monterey Ca.
    93940
649-1000



Virginia Wilson
?
Unknown Can't Find
    99999
?


Type Lower, Upper Bounds:  EOF,,

A>
```

Listing 8-7. (continued)

8.3 An Information management System

The next four sample programs provide a model for an information management system. These programs manage a file of employee names, addresses, wage schedules, and wage reporting mechanisms. Each of these programs is simple, but together they contain all the elements of a more advanced data base management system. They demonstrate the power of the PL/I programming system, while providing the basis for custom application programs.

First, the ENTER program establishes the data base. A second program, called KEYFILE, reads this data base and prepares a key file for direct access to individual records in the data base. A third program, called UPDATE, interacts with the user at the console and allows access to the data base for retrieval and update. Finally, the REPORT program reads the data base to produce a report.

8.3.1 The ENTER Program

Listing 8-8 shows the ENTER program. The ENTER program interacts with the user at the console and constructs the initial data base. The basic input loop between lines 40 and 53 prompts the user for an employee name, age, and hourly wage. ENTER fills the employee data structure with this information. In the example, line 48 fills the address fields with default values defined in the structure on lines 24 through 33. You can terminate the console input by entering EOF.

The employee record contains several fields whose total length is 101 bytes. You can use the $S Compiler switch to verify this value. The OPEN statement on line 37 specifies a fixed record size of 128 bytes, so you can expand the records later. Each record of the emp file holds exactly one employee data structure.

The OPEN statement gives emp the KEYED attribute, and makes each record the fixed size specified in the ENVIRONMENT option. The OPEN statement also specifies the buffer size as 8000 bytes, which PL/I automatically rounds off to 8192 bytes. The program fills each employee record from the console input and writes the record to the employee file named in the command line, with the file type EMP, line 38.

The WRITE statement is in a separate subroutine, named WRITE-IT, starting on line 55. Placing the code in a separate subroutine helps reduce the size of the program because the program calls WRITE-IT at two different points, lines 45 and 52.

Listing 8-9 shows the user interaction with the ENTER program as several employee records are entered. Entering EOF ends the program, closes the file plant1.emp, and records the data on the disk.

```
 1 a   /*****************************************************/
 2 a   /* This program constructs a data base of employee */
 3 a   /* records using a structure declaration.          */
 4 a   /*****************************************************/
 5 a
 6 a ┌─enter:
 7 b │     procedure options(main);
 8 b │     %replace
 9 b │         true  by '1'b,
10 b │          false by '0'b;
11 b │
12 b │     declare
13 b │        1 employee static,
14 b │          2 name      character(30) varying,
15 b │          2 address,
16 b │            3 street character(30) varying,
17 b │            3 city   character(10) varying,
18 b │            3 state  character(12) varying,
19 b │            3 zip    fixed decimal(5),
20 b │          2 age       fixed decimal(3),
21 b │          2 wage      fixed decimal(5,2),
22 b │          2 hours     fixed decimal(5,1);
23 b │
24 b │     declare
25 b │        1 default static,
26 b │          2 street character(30) varying
27 b │            initial('(no street)'),
28 b │          2 city   character(10) varying
29 b │            initial('(no city)'),
30 b │          2 state  character(12)  varying
31 b │            initial('(no state)'),
32 b │          2 zip    fixed decimal(5)
33 b │            initial(00000);
34 b │     declare
35 b │        emp file;
36 b │
37 b │     open file(emp) keyed output environment(f(128),b(8000))
38 b │          title ('$1.EMP');
39 b │
```

Listing 8-8. The ENTER Program

```
40 c     do while(true);
41 c        put list('Employee: ');
42 c        get list(name);
43 c        if name = 'EOF' then
44 d           do;
45 d             call write_it();
46 d             stop;
47 d           end;
48 c        address = default;
49 c        put list ('  Age, Wage: ');
50 c        get list (age,wage);
51 c        hours = 0;
52 c        call write_it();
53 c     end;
54 b
55 b     write_it:
56 c           procedure;
57 c           write file(emp) from(employee);
58 c     end write_it;
59 b
60 b  end enter;
```

Listing 8-8. (continued)

```
A>enter plant1
Employee:  Jackson
  Age, Wage:  25, 6.75
Employee:  Harris
  Age, Wage:  30, 9.00
Employee:  Robinson
  Age, Wage:  41, 15.00
Employee:  Wilson
  Age, Wage:  27, 7.50
Employee:  Smith
  Age, Wage:  25, ,
Employee:  Jones
  Age, Wage:  , ,
Employee:  EOF
A>
```

Listing 8-9. Interaction with the ENTER Program

8.3.2 The KEYFILE Program

Listing 8-10 shows the KEYFILE program, which constructs a key file by reading the data base file created by ENTER. The key file is a sequence of entries consisting of an employee name followed by the key number corresponding to that name. In this case, the key file is also a STREAM file, so you can display it at the console. Line 16 opens the $1.EMP file with the KEYED attribute, specifies each record to be 128 bytes long, and sets a buffer size of 10000 bytes. Line 19 opens the key file named keys as a STREAM file with LINESIZE(60) and a TITLE option that appends KEY as the filetype.

On line 23, the KEYFILE program reads successive records, extracts the key with the KEYTO option, and writes the name and key to both the console and to the key file. The sample interaction in Listing 8-11 illustrates the output from KEYFILE using the plant1.emp data base. Each key value extracted by the READ statement is the relative record number corresponding to the position of the record in the file.

After executing the KEYFILE program, you can use the command

```
A>type plant1.key
```

to display the actual contents of the plant1.key file as shown in Listing 8-12.

```
 1 a   /*********************************************************/
 2 a   /* This program reads an employee record file and       */
 3 a   /* creates another file of keys to access the records. */
 4 a   /*********************************************************/
 5 a
 6 a ┌─keyfile:
 7 b │     procedure options(main);
 8 b │     declare
 9 b │        1 employee static,
10 b │          2 name character(30) varying;
11 b │
12 b │     declare
13 b │        (input, keys) file,
14 b │        k fixed;
15 b │
16 b │     open file(input) keyed environment(f(128),b(10000))
17 b │          title('$1.emp');
18 b │
19 b │     open file(keys) stream output
20 b │          linesize(60) title('$1.key');
21 b │
```

Listing 8-10. The KEYFILE Program

```
22 c     do while('1');
23 c        read file(input) into(employee) keyto(K);
24 c        put skip list(k,name);
25 c        put file(keys) list(name,K);
26 c        if name = 'EOF' then
27 c           stop;
28 c     end;
29 b
30 b  end keyfile;
```

Listing 8-10. (continued)

```
A>keyfile plant1

     0 Jackson
     1 Harris
     2 Robinson
     3 Wilson
     4 Smith
     5 Jones
     6 EOF
A>
```

Listing 8-11. Interaction with the KEYFILE Program

```
A>type plant1.key
'Jackson'  0 'Harris'  1 'Robinson'  2
'Wilson' 3 'Smith' 4 'Jones' 5'EOF
          6
```

Listing 8-12. Contents of the Key File

8.3.3 The UPDATE Program

The UPDATE program in Listing 8-13 allows you to access the data base created by ENTER and indexed through the file created by KEYFILE. The UPDATE program first reads the key file, a STREAM file, into a data structure called keylist. Keylist cross-references the employee name with the corresponding key value in the data base. Lines 20 to 23 declare the data structure that holds these cross-reference values, and lines 37 to 40 fill in the data.

Note: line 39 is not a multiple assignment statement, but rather a definition of a Boolean expression for the variable, eolist.

UPDATE opens the emp file on line 31. The OPEN statement assigns the file the DIRECT attribute, that allows both READ and WRITE operations with the individual records identified by a key value. You then enter an employee name as matchname, and the DO-group between lines 47 and 61 directly accesses the individual records in the data base.

The direct access takes place as follows. Line 48 searches the list of names read from the key file. If there is a match, the READ with KEY statement on line 50 brings the employee record into memory from the emp file. The program displays and updates various fields from the console, and then rewrites the record to the data base with the WRITE with KEYFROM statement on line 58. UPDATE ends execution when you enter an EOF.

Listing 8-14 shows three successive update sessions during which various addresses and work times are updated. In each session, you enter the employee name, access and display the record, and optionally, update the fields. The GET LIST statement is useful here. To change a value, you simply type the new value in the field position. If you do not want to change a value, entering a comma delimiter leaves the field unchanged.

```
 1 a   /*****************************************************/
 2 a   /* This program allows you to retrieve and update    */
 3 a   /* individual records in an employee data base using */
 4 a   /* a keyed file,                                     */
 5 a   /*****************************************************/
 6 a ┌─update:
 7 b │     procedure options(main);
 8 b │     declare
 9 b │         1 employee static,
10 b │           2 name     character(30) varying,
11 b │           2 address,
12 b │             3 street character(30) varying,
13 b │             3 city   character(10) varying,
14 b │             3 state  character(12) varying,
15 b │             3 zip    fixed decimal(5),
16 b │           2 age      fixed decimal(3),
17 b │           2 wage     fixed decimal(5,2),
18 b │           2 hours    fixed decimal(5,1);
19 b │
20 b │     declare
21 b │         1 keylist(100),
22 b │           2 keyname character(30) varying,
23 b │           2 keyval  fixed binary;
24 b │
```

Listing 8-13. The UPDATE Program


```
25 b       declare
26 b           (i, endlist) fixed,
27 b           eolist bit(1) static initial('0'b),
28 b           matchname character(30) varying,
29 b           (emp, keys) file;
30 b
31 b       open file(emp) update direct environment(f(128))
32 b            title ('$1,EMP');
33 b
34 b       open file(keys) stream environment(b(4000))
35 b            title('$1,key');
36 b
37 c     ┌─ do i = 1 to 100 while (^eolist);
38 c     │     get file(keys) list(keyname(i),keyval(i));
39 c     │     eolist = keyname(i) = 'EOF';
40 c     └─ end;
41 b
42 c     ┌─ do while('1'b);
43 c     │     put skip list('Employee: ');
44 c     │     get list(matchname);
45 c     │     if matchname = 'EOF' then
46 c     │        stop;
47 d     │   ┌─ do i = 1 to 100;
48 d     │   │     if matchname = keyname(i) then
49 e     │   │   ┌─ do;
50 e     │   │   │     read file(emp) into(employee)
51 e     │   │   │          key(keyval(i));
52 e     │   │   │     put skip list('Address: ',
53 e     │   │   │         street, city, state, zip);
54 e     │   │   │     put skip list('       ');
55 e     │   │   │     get list(street, city, state, zip);
56 e     │   │   │     put list('Hours:',hours,
57 e     │   │   │     get list(hours);
58 e     │   │   │     write file(emp) from (employee)
59 e     │   │   │           keyfrom(keyval(i));
60 e     │   │   └─ end;
61 d     │   └─ end;
62 c     └─ end;
63 b
64 b   └─ end update;
```

Listing 8-13. (continued)

```
A>update plant1

Employee: Jackson

Address: (no street) (no city) (no state)          0
        '100 W. 3rd St.', 'Fresno', 'Ca.', 93706
Hours:      0.0 : 40.0

Employee: Harris

Address: (no street) (no city) (no state)          0
        '2999 Serra Rd.', 'Chico', 'Ca.', 95926
Hours:      0.0 : 46.0

Employee: EOF

A>update plant1

Employee: Harris

Address: 2999 Serra Rd. Chico Ca. 95926
         ````
Hours: 46.0 : 48.0

Employee: Wilson

Address: (no street) (no city) (no state)   0
         ````
Hours:      0.0 : 35.5

Employee: EOF

A>update plant1

Employee: Wilson

Address: (no street) (no city) (no state)          0
        '556 Palm Ave.', 'Burbank', 'Ca.', 91507
Hours:     35.5 : ,

Employee: EOF
A>
```

**Listing 8-14. Interaction with the UPDATE Program**

### 8.3.4 The REPORT Program

Listing 8-15 shows the REPORT program. The REPORT program uses the updated employee file to produce a list of employees along with their paycheck values. The REPORT program also accesses the employee file, but it reads the file sequentially to produce the desired output. The main DO-group between lines 35 and 51 reads each successive employee record and constructs a title line of the form,

[name]

followed by a dollar amount. REPORT uses the STREAM form of the WRITE statement, lines 41 and 50, to produce the output line. Line 40 includes the embedded control characters ^M and ^J at the end of buff to cause a carriage return and line-feed when writing the buffer. The REPORT program then computes the pay value and assigns it to the CHARACTER-VARYING string called buff, on line 44. In this assignment, PL/I performs an automatic data conversion from FIXED DECIMAL to CHARACTER, with leading blanks. REPORT also scans the leading blanks, replacing them by a dollar sign dash sequence to align the output, and writes the data to the report file.

Listings 8-16 and 8-17 show the output from the REPORT program. In the first case, the command,

```
A>report plant1 $con
```

sends the report to the console for review. In the second case, the command,

```
A>report plant1 plant1.prn
```

sends the output to the disk file plant1.prn. You can then examine the contents of the file with the command:

```
A>type plant1.prn
```

```
 1 a   /************************************************/
 2 a   /* This program reads an employee data base and */
 3 a   /* prints a list of paychecks,                  */
 4 a   /************************************************/
 5 a ┌─report:
 6 b │     procedure options(main);
 7 b │     declare
 8 b │        1 employee static,
 9 b │          2 name      character(30) varying,
10 b │          2 address,
11 b │            3 street character(30) varying,
12 b │            3 city   character(10) varying,
13 b │            3 state  character(12) varying,
14 b │            3 zip    fixed decimal(5),
15 b │          2 age       fixed decimal(3),
16 b │          2 wage      fixed decimal(5,2),
17 b │          2 hours     fixed decimal(5,1);
18 b │
19 b │     declare
20 b │        i fixed,
21 b │        dashes character(15) static initial
22 b │               ('$--------------'),
23 b │        buff character(20) varying,
24 b │        (grosspay, withhold) fixed decimal(7,2),
25 b │        (repfile, empfile) file;
26 b │
27 b │     open file(empfile) keyed environment(f(128),b(4000))
28 b │          title ('$1.EMP');
29 b │     open file(repfile) stream print environment(b(2000))
30 b │          title('$2,$2');
31 b │
32 b │     put list('Set Top of Forms, Press Return');
33 b │     get skip;
34 b │
```

**Listing 8-15. The REPORT Program**

```
35 c     do while('1'b);
36 c        read file(empfile) into(employee);
37 c        if name = 'EOF' then
38 c           stop;
39 c        put file(repfile) skip(2);
40 c        buff = '[' !! name !! ']^m^j';
41 c        write file(repfile) from (buff);
42 c        grosspay = wage * hours;
43 c        withhold = grosspay * .15;
44 c        buff = grosspay - withhold;
45 d        do i = 1 to 15
46 d           while (substr(buff,i,1) = ' ');
47 d        end;
48 c        i = i - 1;
49 c        substr(buff,1,i) = substr(dashes,1,i);
50 c        write file (repfile) from(buff);
51 c     end;
52 b
53 b  end report;
```

**Listing 8-15. (continued)**

```
A>report plant1 $con
Set Top of Forms, Press Return

[Jackson]
$----229.50

[Harris]
$----351.90

[Robinson]
$------0.00

[Wilson]
$----226.32

[Smith]
$------0.00

[Jones]
$------0.00
A>
```

**Listing 8-16. REPORT Generation to the Console**

```
A>report plant1 plant1.prn
Set Top of Forms, Press Return

A>type plant1.prn

[Jackson]
$----229.50

[Harris]
$----351.90

[Robinson]
$------0.00

[Wilson]
$----226.32

[Smith]
$------0.00

[Jones]
$------0.00
```

**Listing 8-17. REPORT Generation to a Disk File**

*End of Section 8*

References: Sections 10.1, 10.8, 11.2, 12 LRM

# Section 9
# Label Constants, Variables, and Parameters

Each of the programs presented so far ends execution either by encountering an end-of-file condition with a corresponding ENDFILE traceback, or by using a special data value that signals the end-of-data condition. The EPOLY program detects the end-of-data condition by checking for the special case where all three input values, x, y, and z, are zero.

Fortunately, PL/I provides more elegant ways to sense the end-of-data condition. In fact, sensing the end-of-data condition is just one of many facilities under the general heading of condition processing. Most often, handling these conditions involves labeled statements. You need some background in label processing before you take up the general topic of condition processing in Section 10.



## 9.1 Labeled Statements

It is an axiom of programming to avoid labeled statements and GOTOs because of the unstructured programs that result. Programs containing many labeled statements are often difficult for other programmers to comprehend. Such programs become unreadable, even to the author, as the program grows in size.

PL/I encourages good structure by providing a comprehensive set of control structures in the form of iterative DO-groups with REPEAT and WHILE options. These control structures preclude the necessity for labeled statements in the general programming schema. You should use these control structures whenever possible, and limit the use of labeled statements to condition processing and locally-defined, computed GOTOs.

Judicious use of labeled statements is appropriate in condition processing. The occurrence of an error, such as a mistyped input data line, is easily handled by transferring program control to a label in an outer block, where recovery takes place. This method of understanding the program flow is simpler than the usual system of flags, tests, and return statements.

## 9.2 Program Labels

Program labels, like other PL/I data types, fall into two broad categories: label constants and label variables. A label constant appears literally within the source program, and its value does not change during program execution. A label variable has no initial value, and you must assign it the value of a label constant through a direct assignment statement, or through the parameter assignments implicit in a sub-routine call.

The following code sequence is an example of a label constant preceding a PL/I statement.

```
on error(1)
   begin;
     put skip list('Bad Input, Try Again');
     goto retry;
   end;
   .
   .
   .
retry: get list(name);
   .
   .
   .
```

The statement `on error(1)` sets a trap for a particular condition. If the condition arises due to an invalid input, then control transfers to the BEGIN block, which outputs an error message, and then transfers control back to the labeled statement. If there is no error on input, control transfers to the next statement following the GET LIST statement.

## 9.3 Computed GOTO

In PL/I, a label constant can contain a single positive or negative literal subscript. A subscripted label constant corresponds to the target of an n-way branch, that is, a computed GOTO. The following code sequence shows a specific example.

```
get list(x);
go to q(x);
q(-1):
      y = f1(x);
      goto endq;
q(0):
     y = f2(x);
     goto endq;
q(2):;
q(3):
     y = f3(x);
endq:
put skip list('f(x)=',y);
     .
     .
     .
```

This code implicitly defines four label constants: q(-1), q(0), q(2), and q(3). The Compiler automatically defines an internal label constant vector,

```
q(-1:3) label constant
```

to hold the values of these label constants.

The preceding statement is not a valid PL/I statement, but indicates what the Compiler does internally when it encounters such statements in the source code. Also, when using such constructs, do not transfer control to a subscript that does not have a corresponding label-constant value. In the preceding case, a branch to q(1) produces undefined results.

## 9.4 Label References

A reference to a label constant can be either local or nonlocal. A local reference to a label constant means that the label occurs as the target of a GOTO statement only in the PROCEDURE or BEGIN block that contains the GOTO. A nonlocal reference to a label constant means that the label occurs on the right side of an assignment to a label variable, as an actual parameter to a subroutine, or as the target of a GOTO statement in an inner nested PROCEDURE or BEGIN block.

Although there is no functional difference between processing a locally-referenced and nonlocally-referenced label constant, a nonlocal reference requires additional space and time. For this reason, PL/I assumes that a subscripted label constant will be only locally referenced. If program control transfers to a subscripted label constant from outside the current environment, undefined results can occur.

As an example, consider the following code sequence:

```
main:
    procedure options(main);
  P1:
    procedure;
      goto lab1;
      goto lab2;
    P2:
      procedure;
        goto lab2;
    end P2;
    lab1:;
    lab2:;
  end P1;
end main;
```

The label constant lab1 is only locally referenced in the procedure P1, while lab2 is the target of both a local reference in P1 and a nonlocal reference in P2.

## 9.5 Example Program

Listing 9-1 shows a nonfunctional program that illustrates the use of various label constants and variables. The label constants in the LABELS program are c(1), c(2), c(3), lab1, and lab2. They are defined by their literal occurrence in the program. The label variables are x, y, z, and g, and are defined by the declarations on lines 10 and 38.

At the start of execution, the label variables have undefined values. The program first assigns the constant value lab1 to the variable x. Label variable y then indirectly receives the constant value lab1 through the assignment on line 12. As a result, all three GOTO statements on lines 14, 15, and 16 are functionally equivalent. Each statement transfers control to the null statement following the label lab1 on line 32.

The subroutine call on line 18 shows a different form of variable assignment. Lab2 is an actual parameter sent to the procedure P, and assigned to the formal label variable g. In this program, the subroutine call transfers program control directly to the statement labeled lab1.

The DO-group beginning on line 20 initializes the variable label vector z to the corresponding constant label vector values of c. Due to this initialization, the two computed GOTO statements, starting on line 25, have exactly the same effect.

```
 1 a   /*****************************************************/
 2 a   /* This is a nonfunctional program.  Its purpose is */
 3 a   /* to illustrate the concept of label constants and */
 4 a   /* variables.                                        */
 5 a   /*****************************************************/
 6 a ┌─Labels:
 7 b │       procedure options(main);
 8 b │       declare
 9 b │         i fixed,
10 b │         (x, y, z(3)) label;
11 b │       x = lab1;
12 b │       y = x;
13 b │
14 b │       goto lab1;
15 b │       goto x;
16 b │       goto y;
17 b │
18 b │       call P(lab2);
19 b │
```

**Listing 9-1. An Illustration of Label Variables and Constants**

```
20 c        do i = 1 to 3;
21 c           z(i) = c(i);
22 c        end;
23 b
24 b        i = 2;
25 b        goto z(i);
26 b        goto c(i);
27 b
28 b        c(1):;
29 b        c(2):;
30 b        c(3):;
31 b
32 b        lab1:;
33 b        lab2:;
34 b
35 b        P:
36 c          procedure (g);
37 c          declare
38 c            g label;
39 c          goto g;
40 c        end P;
41 b
42 b  end Labels;
```

**Listing 9-1. (continued)**

*End of Section 9*

References: Sections 3.3, 8.5 LRM

# Section 10
# Condition Processing

Condition processing is an important facility of any production programming language. The language should allow a program to intercept and handle run-time error conditions with program-defined actions, and then continue execution.

For example, a common condition occurs when a program is reading input data from an interactive console, and you inadvertently enter a value that does not conform to the data type of the input variable. The PL/I run-time system signals a conversion error, and in the absence of any program-defined action, ends program execution with a traceback. If this premature termination occurs after hours of data entry, it causes a considerable amount of wasted effort. This is unacceptable in a production environment.



## 10.1 Condition Categories

PL/I provides nine categories of conditions. They are:

- ERROR
- FIXEDOVERFLOW
- OVERFLOW
- UNDERFLOW
- ZERODIVIDE
- ENDFILE
- UNDEFINEDFILE
- KEY
- ENDPAGE

The first five categories include all arithmetic error conditions and miscellaneous conditions that can arise during I/O setup and processing. They also include conversion errors between the various data types. The last four categories apply to a specific file that the run-time I/O system is accessing. Each condition has an associated subcode that provides information about the source of the condition.

## 10.2 Condition Processing Statements

The ON, REVERT, and SIGNAL statements implement condition processing in PL/I. The ON statement defines the actions that take place upon encountering a condition. The REVERT statement disables the ON statement, and recovers any previously stacked condition. The SIGNAL statement allows your program to signal various conditions.

### 10.2.1 ON and REVERT

The following code sequence illustrates the ON and REVERT statements inside a DO-group.

```
do while('1'b);
    on endfile(sysin)
    EOF = '1'b;
        .
        .
        .
    revert endfile(sysin);
end;
```

Here, both the ON and the REVERT statement execute on each iteration. Processing the ON and REVERT statements involves run-time overhead. To avoid this, code the same DO-group as follows:

```
on endfile(sysin)
EOF = '1'b;
do while('1'b);
    .
    .
    .
end;
```

PL/I automatically executes the REVERT statement for any ON conditions that you enable inside a procedure block when control passes outside the block. The program shown in Listing 10-1 illustrates this concept.

```
 1 a   /*****************************************************/
 2 a   /* This program is nonfunctional.  Its purpose is to */
 3 a   /* illustrate how PL/I executes the ON and REVERT    */
 4 a   /* statements.                                       */
 5 a   /*****************************************************/
 6 a   auto_revert:
 7 b       procedure options(main);
 8 b       declare
 9 b         i fixed,
10 b         sysin file;
11 b
12 c       do i = 1 to 10000;
13 c          call P(i,exit);
14 c          exit:
15 c       end;
16 b
17 b       P:
18 c        procedure (index,lab);
19 c        declare
20 c          (t, index) fixed,
21 c          lab label;
22 c
23 c        on endfile(sysin)
24 c           goto lab;
25 c
26 c        put skip list(index,':');
27 c        get list(t);
28 c        if t = index then
29 c        goto lab;
30 c        end P; /* implicit REVERT supplied here */
31 b
32 b   end auto_revert;
```

**Listing 10-1. The REVERT Program**

In the REVERT program, line 13 calls the procedure P and passes to it the actual parameters i, the DO-group index, and the label constant exit. The ON statement inside P executes every time the procedure is called. If PL/I did not supply the REVERT statement automatically, the Condition Stack would overflow when the value of the index count reached 17. Thus, REVERT has three possible ways to exit the procedure P.

If you enter an end-of-file character, CTRL-Z, REVERT executes the enabled ON condition and sends control through the label variable lab to the statement labeled exit. PL/I deactivates the procedure and executes the REVERT statement because the GOTO statement transfers control outside the environment of P.

The second possible exit follows the test on line 28. If you enter a value equal to the index, then the GOTO statement on line 29 executes and again sends control outside the environment of P.

Finally, if control reaches the end of P, PL/I executes the REVERT statement and disables the ON condition set on line 23. No matter how control leaves the environment of the procedure, PL/I always disables the ON condition.

### 10.2.2 SIGNAL

The SIGNAL statement activates the ON-body, the body of statements corresponding to a particular ON statement. Thus, processing a SIGNAL statement has the same effect as when the run-time system signals the condition.

The following code sequence illustrates the SIGNAL statement.

```
on endfile(sysin)
   stop;

do while('1'b);
   get list(buff);
   if buff = 'END' then
      signal endfile(sysin);
   put skip list(buff);
end;
   .
   .
   .
```

This code executes the SIGNAL statement whenever the GET LIST statement reads the value END from the file SYSIN. Thus, the ON condition receives control on a real end-of-file, or when the value END is read.

## 10.3 Examples of Condition Processing

The following two programs, FLTPOLY2 and COPYLPT, incorporate some condition processing, so you can see how these concepts are implemented.

### 10.3.1 The FLTPOLY2 Program

Listing 10-2 shows the FLTPOLY2 program. This is essentially the same program listed in Section 7-1. The only difference is that it incorporates condition processing to intercept the end-of-file condition for the file SYSIN. If you run this program, you will see how you can end execution with a CTRL-Z character. Unlike FLTPOLY, if you enter all zeros, FLTPOLY2 simply evaluates the polynomial and prompts you for more input.

```
 1 a   /*******************************************************/
 2 a   /* This program evaluates a polynomial expression      */
 3 a   /* using FLOAT BINARY data.  It also traps the end-of- */
 4 a   /* file condition for the file SYSIN.                  */
 5 a   /*******************************************************/
 6 a   fltpoly2:
 7 b       procedure options(main);
 8 b       %replace
 9 b           false by '0'b,
10 b           true  by '1'b;
11 b       declare
12 b         (x,y,z) float binary(24),
13 b         eofile bit(1) static initial(false),
14 b         sysin file;
15 b
16 b       on endfile(sysin)
17 b          eofile = true;
18 b
19 c       do while(true);
20 c          put skip(2) list('Type x,y,z: ');
21 c          get list(x,y,z);
22 c
23 c          if eofile then
24 c             stop;
25 c
26 c          put skip list('        2');
27 c          put skip list('       x  + 2y + z =',P(x,y,z));
28 c       end;
29 b
30 b       P:
31 c        procedure (x,y,z) returns (float binary(24));
32 c        declare
33 c          (x,y,z) float binary(24);
34 c        return (x * x + 2 * y + z);
35 c       end P;
36 b
37 b   end fltpoly2;
```

**Listing 10-2. The FLTPOLY2 Program**

### 10.3.2 The COPYLPT Program

Listing 10-3 shows an example of I/O processing using ON conditions. The COPYLPT program copies a STREAM file from the disk to a PRINT file, while properly formatting the output line with a page header and line numbers. The program accepts console input to obtain the parameters for the copy operation, and provides error exits and retry operations for each input value. COPYLPT sets up various ON-units to intercept errors during the copy operation that takes place in the iterative DO-group between lines 71 and 76. The following sections discuss the individual parts of the program.

```
 1 a   /*************************************************/
 2 a   /* This program copies a STREAM file on disk to a */
 3 a   /* PRINT file, and formats the output with a page */
 4 a   /* header, and line numbers.                      */
 5 a   /*************************************************/
 6 b   copy: procedure options(main);
 7 b
 8 b         declare
 9 b           (sysin, sourcefile, printfile) file,
10 b           (pagesize, pagewidth, spaces, linenumber) fixed,
11 b           (line character(14), buff character(254)) varying;
12 b
13 b         put list('^z    File to Print Copy Program');
14 b
15 b         on endfile(sysin)
16 b            go to typeover;
17 b
18 b         typeover:
19 b            put skip(5) list(How Many Lines Per Page?   ');
20 b            get list(pagesize);
21 b
22 b            put skip list('How Many Column Positions? ');
23 b            get skip list(pagewidth);
24 b
25 b         on error(1)
26 c         ┌ begin;
27 c         │   put list('Invalid Number, Type Integer');
28 c         │   go to getnumber;
29 c         └ end;
30 b         getnumber:
31 b            put skip list('Line Spacing (1=Single)?   ');
32 b            get skip list(spaces);
33 b            revert error(1);
34 b
35 b         put skip list('Destination  Device/File:  ');
36 b         get skip list(line);
37 b
```

**Listing 10-3. The COPYLPT Program**

```
38 b          open file(printfile) print pagesize(pagesize)
39 b               linesize(pagewidth) title(line);
40 b
41 b          on undefinedfile(sourcefile)
42 c           begin;
43 c             put skip list('"',line,'" isn''t a Valid Name');
44 c             go to retry;
45 c           end;
46 b          retry:
47 b             put skip list('Source File to Print?       ');
48 b             get list(line);
49 b             open file(sourcefile) stream  environment(b(8000))
50 b                  title(line);
51 b          on endfile(sourcefile)
52 c           begin;
53 c             put file(printfile) page;
54 c             stop;
55 c           end;
56 b
57 b          on endfile(printfile)
58 c           begin;
59 c             put skip list('^g^g^g^g Disk is Full');
60 c             stop;
61 c           end;
62 b
63 b          on endpage(printfile)
64 c           begin;
65 c             put file(printfile) page skip(2)
66 c                 list('PAGE',pageno(printfile));
67 c             put file(printfile) skip(4);
68 c           end;
69 b
70 b          signal endpage(printfile);
71 c        do linenumber = 1 repeat(linenumber + 1);
72 c           get file (sourcefile) edit(buff) (a);
73 c           put file (printfile)
74 c               edit(linenumber,'|',buff) (f(5),x(1),a(2),a);
75 c           put file (printfile) skip(spaces);
76 c        end;
77 b
78 b   end copy;
```

**Listing 10-3. (continued)**

The COPYLPT program begins by reading five values:

- the number of lines on each page
- the width of the printer line

- the line spacing, normally single- or double-spaced output
- the destination file or device
- the source file or device

While entering these parameters, you can type an end-of-file CTRL-Z character and restart the prompting.

The PUT LIST statement on line 13 writes the initial sign-on message. Recall that PL/I allows control characters in string constants. Here, the first character of the message is a CTRL-Z, which clears the screen if you are using an ADM-3A™ CRT device. If you are using some other device, you can substitute the proper character and recompile the program.

The ON statement of line 15 traps the ENDFILE condition for the file SYSIN, so that execution begins at typeover whenever the console reads an end-of-file character.

Lines 19 through 23 read the first two parameters with no error checking other than detecting the end-of-file. Line 25 however, intercepts conversion errors for all operations that follow. If the GET statement on line 32 reads a nonnumeric field, control passes to the on-body between lines 26 and 29 that writes an error message, branches to getnumber, and retries the input operation. Following successful input of the parameter spaces, the REVERT statement on line 33 disables the conversion error handling.

COPYLPT opens the input and output files between lines 38 and 50. The program assumes that the output file can always be opened, but detects an UNDEFINED input file, so you can correct the filename.

The program executes two ON ENDFILE statements between lines 51 and 61. The first statement traps the input end-of-file condition and performs a page eject on the output file. This ensures that the printer output is at the top of a new page after completing the print operation. The STOP statement included in this ON-unit completes the processing with an exit.

The second ON-unit intercepts the end-of-file condition on the print file. This can only occur if the disk file fills, so the unit prints the message,

```
Disk is Full
```

and ends execution. The CTRL-G character sends a series of beeps to the CRT as an alarm. The run-time system closes all files upon termination, so that the print file is intact to the full capacity of the disk.

Line 63 begins an ON ENDPAGE unit that intercepts the end-of-page condition for the print file. Whenever the run-time system signals this condition, the ON-unit moves to the top of the next page, skips two lines, prints the page number, and skips four more lines before returning to the signal source. The SIGNAL statement on line 70 starts the print file output on a new page by sending control to the ON-unit defined on line 63. All subsequent ENDPAGE signals are generated by the run-time system at the end of each page.

The DO-group beginning on line 71 initializes and increments a line counter on each iteration. The GET EDIT statement on line 72 specifies an A, alphanumeric, format. This fills the buffer with the next input line up to, but not including, the carriage return line-feed sequence. The PUT EDIT statement on line 73 writes the line to the destination file with a preceding line number, a blank, a vertical bar, and another blank, resulting from the A(2) field. If the run-time system signals the ENDPAGE condition while executing the PUT statement on line 75, the format item SKIP(spaces) might not be processed.

Listing 10-4 shows the user interaction with the COPYLPT program. Here, the source file is the LABELS.PLI program, and $LST, the physical printer, is the destination.

```
A>copylpt
    File to Print Copy Program


How Many Lines Per Page?      26

How Many Column Positions?    80

Line Spacing (1=Single)?      Yes
Invalid Number, Type Integer
Line Spacing (1=Single)?      1

Destination Device/File:      $lst

Source File to Print?         copy.pil

" copy.pil " isn't a Valid Name
Source File to Print?         copy.pli
```

**Listing 10-4. Interaction with COPYLPT**

Listing 10-5 shows two pages of output produced by the program.

```
PAGE           1


   1 ¦  /*************************************************/
   2 ¦  /* This program copies one file to another using */
   3 ¦  /* buffered I/O.                                 */
   4 ¦  /*************************************************/
   5 ¦  copy:
   6 ¦      procedure options(main);
   7 ¦      declare
   8 ¦         (input_file,output_file) file;
   9 ¦
  10 ¦      open file (input_file) stream
  11 ¦           environment(b(8192)) title('$1,$1');
  12 ¦
  13 ¦      open file (output_file) stream output
  14 ¦           environment(b(8192)) title('$2,$2');
  15 ¦      declare
  16 ¦         buff character(254) varying;
  17 ¦
  18 ¦      do while('1'b);
  19 ¦         read  file (input_file)  into (buff);
  20 ¦         write file (output_file) from (buff);


PAGE           2


  21 ¦      end;
  22 ¦  end copy;
```

**Listing 10-5. Output from COPYLPT**

This example shows that you can incorporate error handling in your programs to make them easier to use. In fact, you could enhance the COPYLPT program to handle errors in the first two input lines, or errors in the destination filename.

To gain further experience, you could go back over all the previous examples and add ON-units to trap invalid input data and end-of-file conditions. Modifying these small programs gives you a good foundation in condition processing.

*End of Section 10*

References: Section 9 LRM

# Section 11
# Character String Processing

PL/I provides powerful character-string handling capabilities essential in a commercial production language. This section presents two sample programs that illustrate the use of some PL/I character-string functions. After you read the text and study the sample programs, you can make changes in the programs to expand your knowledge of PL/I.

## 11.1 The OPTIMIST Program

Our first example of string processing is a program called the OPTIMIST. The OPTIMIST program turns a negative sentence into a positive sentence. The OPTIMIST performs this task by using the character-string facilities of PL/I.



Listing 11-1 shows the OPTIMIST program. The first segment, between lines 12 and 23, defines the data items used in the program. The remaining portion reads a sentence from the console, ending with a period, and retypes the sentence in its positive form. Listing 11-2 shows a sample console interaction with the OPTIMIST. The OPTIMIST works well if sentences are simple, but complicated sentences confuse the program.

Line 13 gives the OPTIMIST vocabulary of negative words, with the corresponding positive words on line 15. Thus, never becomes always, and none becomes all. OPTIMIST replaces the word not with an empty string. Lines 17 through 20 declare the upper- and lower-case alphabets for case translation in the sentence processing section.

OPTIMIST constructs each successive input sentence between lines 28 and 32, where the DO-group reads another word, and concatenates the word on the end of the sentence. The SUBSTR test in the DO WHILE heading checks for a period at the end.

**Note:** OPTIMIST can only accept a sentence whose maximum length is 254 characters. PL/I discards any additional characters.

After reading the complete sentence, OPTIMIST translates all upper-case characters to lower-case to scan the negative words. It performs this case translation on line 33 by using the built-in TRANSLATE function. OPTIMIST uses the built-in VERIFY function on line 34 to ensure that the sentence consists only of letters and a period.

If the sentence consists of characters other than letters or a period, the VERIFY function returns the first nonzero position that does not match, and the OPTIMIST responds with:

```
Actually, that's an interesting idea.
```

If the VERIFY function returns a zero value, then the sentence contains only translated lower-case letters and a period. In this case, control transfers to the DO-group between lines 36 and 42. On each iteration, OPTIMIST uses the built-in INDEX function to search for the next negative word, given by negative (i). If found, it sets j to the position of the negative word, and in the assignment statement on line 39, replaces it with the corresponding positive word. In this assignment, the portion of the sentence that occurs before the negative word is given by,

```
substr(sent,1,j-1)
```

while the replacement value for the negative word is given by,

```
positive(i)
```

and the portion of the sentence that follows the negative word being replaced is given by:

```
substr(sent,j+length(negative(i)))
```

The OPTIMIST concatenates these three segments to produce a new sentence with the negative word replaced by the positive word. It then sends the resulting sentence to the console, and loops back to read another input. Because all negative words have a leading blank, the negative portion is always found at the beginning of a word. Thus, OPTIMIST replaces nevermind with alwaysmind. This can produce interesting results.

You could make at least three improvements to the OPTIMIST. First, if the sentence exceeds 254 characters, the input scan never stops, because the period is not found. You could include a check to ensure that the newly appended word does not exceed the maximum size.

Second, there is no condition processing in the DO-group between lines 25 and 45, so the OPTIMIST never stops talking. It ends only through input of a CTRL-Z, end-of-file, or CTRL-C, system warm start. You could include an ON-unit to detect an end-of-file to end the program in a reasonable fashion.

Finally, you could try to make the OPTIMIST smarter!

```
 1 a   /*****************************************************/
 2 a   /* This program demonstrates PL/I character string   */
 3 a   /* processing by turning a negative sentence into a  */
 4 a   /* positive one.                                     */
 5 a   /*****************************************************/
 6 a   optimist:
 7 b       procedure options(main);
 8 b       %replace
 9 b           true   by '1'b,
10 b           false  by '0'b,
11 b           nwords by 5;
12 b       declare
13 b           negative (1:nwords) character(8) varying static initial
14 b               (' never',' none',' nothing',' not',' no'),
15 b           positive (1:nwords) character(10) varying static initial
16 b               (' always',' all',' something',' ',' some'),
17 b           upper character(28) static initial
18 b               ('ABCDEFGHIJKLMNOPQRSTUVWXYZ. '),
19 b           lower character(28) static initial
20 b               ('abcdefghijklmnopqrstuvwxyz. '),
21 b           sent character(254) varying,
22 b           word character(32)  varying,
23 b           (i,j) fixed;
24 b
25 c       do while(true);
26 c          put skip list('What''s up? ');
27 c          sent = ' ';
28 d          do while
29 d             (substr(sent,length(sent)) ^= '.');
30 d             get list (word);
31 d             sent = sent !! ' ' !! word;
32 d          end;
33 c          sent = translate(sent,lower,upper);
34 c          if verify(sent,lower) ^= 0 then
35 c             sent = ' that''is an interesting idea.';
36 d          do i = 1 to nwords;
37 d             j = index(sent,negative(i));
38 d             if j ^= 0 then
39 d                sent = substr(sent,1,j-1) !!
40 d             positive(i) !!
41 d             substr(sent,j+length(negative(i)));
42 d          end;
43 c          put list('Actually,'!!sent);
44 c          put skip;
45 c       end;
46 b
47 b   end optimist;
```

**Listing 11-1. The OPTIMIST Program**

```
A>optimist

What's up? Nothing is up.
Actually, something is up.

What's up? This is not fun.
Actually, this is fun.

What's up? Programs like this never make sense.
Actually, programs like this always make sense.

What's up? Nothing is easy that is not complicated.
Actually, something is easy that is complicated.

What's up? Nobody cares and its none of your business.
Actually, somebody cares and its all of your business.

What's up? The price of everything.
Actually, the price of everything.

What's up? Boy are you stupid.
Actually, boy are you stupid.

What's up? Dont get smart with me.
Actually, dont get smart with me.

What's up? You started it I didnt.
Actually, you started it i didnt.

What's up? No I did not.
Actually, some i did.

What's up? Thats better.
Actually, thats better.

What's up? You are hard to talk to.
Actually, you are hard to talk to.

What's up? There you go again.
Actually, there you go again.

What's up? Thats it I quit.
Actually, thats it i quit.
```

**Listing 11-2. Interaction with the OPTIMIST**

```
What's up? Stop that.
Actually, stop that.

What's up? If you dont stop I will pull your plug.
Actually, if you dont stop i will pull your plug.

What's up? You can not pull my plug.
Actually, you can pull my plug.

What's up? I know.
Actually, i know.

What's up? ^Z

END OF FILE (1), File: SYSIN=CON
Traceback: 09C5 0970 0157 4100 # 0909 0529 8090 0157
A>
```

**Listing 11-2. (continued)**

## 11.2 A Parse Function

This section presents a more practical application of string processing. It is often useful to have a separate subroutine in a program that reads a line of input and separates it into individual numbers and characters. Such a subroutine is called a parser, or a free-field scanner. The FSCAN program, shown in Listing 11-3, gives an example of a parser.

FSCAN demonstrates the embedded subroutine called GNT, Get Next Token, which parses an input line into separate items called tokens. Once you test GNT, you can extract it from this program and put it into a production program where required. Section 13.4 uses GNT to compute values of arithmetic expressions.

Listing 11-4 shows interaction with the FSCAN program. FSCAN reads a line of input, parses the line into separate tokens, and then writes the tokens back to the console, with surrounding apostrophes. The tokens are just numeric values, such as 1234.56, or individual letters and special characters. GNT bypasses all intervening blanks between the tokens in the token scan.

The FSCAN program has three parts. The first part, lines 10 to 12, defines the global data area called token, used by the GNT procedure. The second part, lines 14 to 42, is the GNT procedure itself. The third part is the DO-group between lines 44 and 47 that performs the test of the GNT function procedure.

```
 1 a   /************************************************/
 2 a   /* This program tests the procedure called GNT, a */
 3 a   /* free-field scanner (parser) that reads a line  */
 4 a   /* of input and breaks it into individual parts.  */
 5 a   /************************************************/
 6 a ┌─fscan:
 7 b │     procedure options(main);
 8 b │     %replace
 9 b │         true by '1'b;
10 b │     declare
11 b │         token character(80) varying
12 b │         static initial('');
13 b │
14 b │   ┌─ gnt:
15 c │   │     procedure;
16 c │   │     declare
17 c │   │         i fixed,
18 c │   │         line character(80) varying
19 c │   │         static initial('');
20 c │   │
21 c │   │     line = substr(line,length(token)+1);
22 d │   │   ┌─do while(true);
23 d │   │   │    if line = '' then
24 d │   │   │       get edit(line) (a);
25 d │   │   │    i = verify(line,' ');
26 d │   │   │    if i = 0 then
27 d │   │   │       line = '';
28 d │   │   │    else
29 e │   │   │   ┌─do;
30 e │   │   │   │   line = substr(line,i);
31 e │   │   │   │   i = verify(line,'0123456789.');
32 e │   │   │   │ ┌─if i = 0 then
33 e │   │   │   │ │     token = line;
34 e │   │   │   │ └─else
35 e │   │   │   │ ┌─if i = 1 then
36 e │   │   │   │ │     token = substr(line,1,1);
37 e │   │   │   │ └─else
38 e │   │   │   │       token = substr(line,1,i-1);
39 e │   │   │   │   return;
40 e │   │   │   └─end;
41 d │   │   └─end;
42 c │   └─ end gnt;
43 b │
```

**Listing 11-3. The FSCAN Program**

```
44 c       do while(true);
45 c          call gnt;
46 c          put edit('''''!!token!!''''') (x(1),a);
47 c       end;
48 b
49 b    end fscan;
```

**Listing 11-3. (continued)**

```
A>fscan
  88+9.9
 '88' '+' '9.9'
  1234567    89.10
 '1234567' '89.10'
  1,2,3,4,5,6,7
 '1' ',' '2' ',' '3' ',' '4' ',''5' ',' '6' ',' '7'
  ,,,,666,,,, 7.7.7.
 ',,,,666,,,,' '7.7.7.'
^Z

End of File (7), File: SYSIN=CON
Traceback: 0B6E 27CA 0143 00FF # 0B78 0986 0143 01F5
A>
```

**Listing 11-4. Interaction with the FSCAN Program**

### 11.2.1 The GNT Procedure

GNT stores the input line in the character variable called line that is initially empty due to the declaration on line 18. On the first call, GNT extracts the first portion of line and places it in token, which becomes the next input item. On each successive call, GNT removes the previous token value from the beginning of a line before scanning the next item.

For example, suppose the input line is,

␣␣␣88*9.9

where ␣ represents a blank character. On the first call to GNT, both token and line are empty strings. The assignment on line 21 removes the previous value of token and leaves line as an empty string. The DO-group between lines 22 and 41 ensures that the line buffer is always filled. If GNT encounters an empty buffer, the GET EDIT statement, line 24, immediately refills it. The call to the built-in VERIFY function on line 25 returns the first position in line that is not blank.

If VERIFY returns a 0, then the entire line is blank and must be cleared. The refill operation takes place on the next iteration. If the line is not entirely blank, then control transfers to the DO-group beginning on line 29.

### 11.2.2 The DO-Group

Processing in the DO-group takes place as follows. On entry, the value of i is the first nonblank position of the line buffer. Thus, the statement on line 30 removes the preceding blanks from line, so the next token starts at the first position. GNT then calls the VERIFY function to determine if the next item in line is a number.

The assignment statement on line 31 sets i to 0 if the entire buffer consists of numbers and decimal points. Line 31 sets i to 1 if the first item is not a number or a period. It sets i to a larger value than 1 if the first item is a number that does not extend through the entire line buffer. Thus, this sequence of tests, starting at line 32, either extracts the entire line (i = 0), the first character of the line (i = 1), or the first portion of the line (i >1).

In the preceding example input line, on the first iteration GNT sets line to,

```
b  b  b  8  8  *  9  .  9
1  2  3  4  5  6  7  8  9
```

where the index 1 through 9, in line, is shown below each character. On line 30, GNT removes the initial blanks, leaving line as:

```
8  8  *  9  .  9
1  2  3  4  5  6
```

Line 31 calls the VERIFY function that locates the first position containing a nondigit or period character. In this case, VERIFY returns the value 3, which corresponds to the * in position 3. As a result of the tests, FSCAN executes line 38 and produces the equivalent of:

```
substr('88*9.9',1,2)
```

This results in a token value of 88, which is the next number in line.

On the next call, GNT removes token from line using the SUBSTR operation on line 21 and leaves line as:

```
*   9   .   9

1   2   3   4
```

The VERIFY function on line 31 returns the value 1, because the leading position of line is not a digit or a period. Line 36 extracts and returns the first character of line as the value of token.

The third call to GNT gets the last token in line by first extracting the *. This leaves line as:

```
9   .   9

1   2   3
```

This time, because all characters are either digits or periods, the VERIFY function returns a 0 and GNT executes line 33. This results in a token value of 9.9, which is the remainder of line.

The fourth call to GNT clears the previous value of token from line, so that line is the empty string. This causes GNT to execute the GET EDIT statement, line 24, and refill line from the console. Execution proceeds in this manner until you stop the program with a CTRL-Z or CTRL-C input.

This simple parser has some obvious flaws. It does not trap end-of-file conditions. You could include an ON-unit to detect this condition, and return a null token value to indicate there is no more input. Furthermore, GNT does not detect multiple period characters. This would cause a subsequent conversion signal (ERROR(1)) if you attempt to convert to a decimal value.

These enhancements give you an improved version of GNT that you can incorporate into any of your programs.

*End of Section 11*

References: Sections 3.2, 6.4, 6.8 LRM

# Section 12
# List Processing

For some programs it is difficult to determine the exact memory requirements before the program runs. List processing is an example of this kind of program because the number of data elements can vary considerably while the program is running.

PL/I has subroutines in the Run-time Subroutine Library (RSL) that dynamically manage storage allocation. When the operating system loads a PL/I program into the Transient Program Area (TPA) or partition, PL/I first initializes all the remaining free memory as a linked list. The list elements contain information fields and pointers to other list elements. A program dynamically allocates memory by using the ALLOCATE statement and releases memory using the FREE statement. PL/I continuously keeps all memory segments connected to one another by using the linked-list mechanism.

The programs in this section illustrate list processing in two cases where it is not easy to predetermine the amount of storage required.



## 12.1 Based and Pointer Variables

You can visualize a based variable as a template that fits over a region of memory but has no storage directly allocated to it. A pointer variable is just a two-byte value that holds the address of a variable. When you use a pointer variable, you are programmatically placing this based variable template over a particular piece of memory. The method depends on the form of the based variable declaration.

If the based variable declaration does not include an implied base, then you must qualify any reference to the based variable with a pointer. If the based variable declaration includes an implied base, then you can include a pointer qualifier in any reference to the based variable, or you can simply use the implied pointer given in the declaration as a base.

Consider the following example declaration:

```
declare
  i fixed,
  mat(0:5) fixed,
  (p, q) pointer,
  x fixed based,
  y fixed based(p),
  z fixed based(f());
```

PL/I allocates storage for the two variables i and mat because they are not based variables. PL/I also assigns storage locations for the two pointer variables p and q. However, the three variables x, y, and z are declared as based variables, and they have no storage locations prior to execution. Instead, PL/I determines their actual storage addresses as the program runs. The variable x has no implied base, so every reference to x must have a pointer qualifier such as:

```
p->x = 5;
```

or,

```
q->x = 6;
```

The first statement assigns the value 5 to the fixed two-byte variable at the memory location given by p. The second statement assigns the value 6 to the location given by q.

The variable y, on the other hand, has an implied base that means you can reference it with or without a pointer qualifier. The reference

```
y = 5;
```

equals

```
p->y = 5;
```

and thus,

```
y = 5;    and    q->y = 6;
```

have exactly the same effect as the two preceding assignments to x.

The variable z, like the variable y, has an implied base. In this case, the base is an invocation of a pointer-valued function with no arguments. For example, the function f can take the form:

```
f:
   procedure returns(pointer);
   return (addr(mat(i)));
end f;
```

Using this definition of f, you can reference z as:

```
p->z = 5;
```

or,

```
z = 6;
```

The first form is equivalent to those shown above, with the location derived from the pointer variable p. The second form however, is an abbreviation for:

```
f() -> z = 6;
```

In this case, PL/I evaluates the function f to produce the storage address for the based variable z. This form has a twofold advantage. First, the pointer-valued expression can be complex, and not restricted to a simple pointer variable. Second, the code for function f appears only once, rather than being duplicated at each variable reference. This can save a considerable amount of space in a program.

**Note:** the implied base must be in the scope of the declaration for the based variable.

The following *incorrect* code sequence illustrates this concept:

```
main:
     procedure options(main);
     declare
        x based(p),
        y based(q),
        p pointer;
        begin;
          declare
            (p,q) pointer;
          x = 5;
          y = 10;
        end;
          declare
            q pointer;
end main;
```

Because the variables x and y are based on p and q, the pointers p and q must be in the same or encompassing scope. Here the pointers p and q are declared in the embedded BEGIN block that is a different environment.

## 12.2 The REVERSE Program

Our first example of list processing is a program called REVERSE. The OPTIMIST program in Section 11 can accept a sentence with a maximum of 254 characters, the maximum string length. REVERSE, however, accepts sentences of virtually any length by using a list structure instead of a single character string. Instead of performing word substitution, REVERSE simply reverses the input sentence.

Listing 12-1 shows the REVERSE program, which is divided into three parts. The first part, lines 12 through 17, reads a sentence from the console and writes the sentence back to the console in reverse order. Each input sentence consists of a sequence of words up to 35 characters in length. This is sufficient to hold,

supercalifragilisticexpialidocious

one of the longest words in the English language.

To simplify the input processing, REVERSE requires a space before the period that ends the sentence. REVERSE also ends execution when you type an empty sentence.

The second part of REVERSE is a separate subroutine, called read_it, which starts on line 19. The third part is a subroutine called write_it, which begins on line 37. Making these functions separate subroutines in the main program simplifies the overall structure.

Listing 12-2 shows the console interaction with REVERSE.

```
 1 a   /************************************************/
 2 a   /* This program reads a sentence and reverses it. */
 3 a   /************************************************/
 4 a   reverse:
 5 b       procedure options(main);
 6 b       declare
 7 b           sentence pointer,
 8 b           1 wordnode based (sentence),
 9 b             2 word character(35) varying,
10 b             2 next pointer;
11 b
12 c       do while('1'b);
13 c          call read_it();
14 c          if sentence = null then
15 c             stop;
16 c          call write_it();
17 c       end;
18 b
19 b       read_it:
20 c           procedure;
21 c           declare
22 c             newword character(35) varying,
23 c             newnode pointer;
24 c           sentence = null;
25 c           put skip list('What''s up? ');
26 d           do while('1'b);
27 d              get list(newword);
28 d              if newword = '.' then
29 d                 return;
30 d              allocate wordnode set (newnode);
31 d              newnode->next = sentence;
32 d              sentence      = newnode;
33 d              word          = newword;
34 d           end;
35 c       end read_it;
36 b
```

**Listing 12-1. The REVERSE Program**

```
37 b     write_it:
38 c         procedure;
39 c         declare
40 c           p pointer;
41 c         put skip list('Actually, ');
42 d         do while (sentence ^= null);
43 d            put list(word);
44 d            p = sentence;
45 d            sentence = next;
46 d            free p->wordnode;
47 d         end;
48 c         put list('.');
49 c         put skip;
50 c     end write_it;
51 b
52 b end reverse;
```

**Listing 12-1. (continued)**

```
A>reverse

What's up? North is up .

Actually, up is North .

What's up? The rain in Spain falls mainly in the plain .

Actually, plain the in mainly falls Spain in rain The .

What's up? 3 + 5 = 8 .

Actually, 8 = 5 + 3 .

What's up? .

A>
```

**Listing 12-2. Interaction with the REVERSE Program**

The REVERSE program stores each word in a separate area of memory, obtained using the ALLOCATE statement on line 30. On each iteration of the DO-group, the ALLOCATE statement obtains a unique section of the free memory space sufficiently large to hold the wordnode structure defined on line 8. The wordnode elements are linked together through the next field of each allocation, and the beginning of the list is given by the value of the sentence pointer variable.

Each allocation consumes 38 bytes. You can verify this by examining the Symbol Table. The wordnode structure is 38 bytes long because word is declared as CHARACTER(35) VARYING, and requires one byte to hold the current length, 35 bytes to hold the string itself, and is followed by a two-byte pointer value.

For example, given the input sentence,

```
I SHALL RETURN .
```

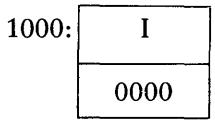
REVERSE executes the ALLOCATE statement three times, once for each word in the list.

Suppose that these three memory allocations are found at addresses 1000, 2000, and 3000. The REVERSE program begins by reading the sentence in the main DO-group in the read_it procedure. It initializes the sentence pointer to the null address (0000). Upon entering the DO-group at line 26, the value of sentence appears as follows:

SENTENCE: 0000

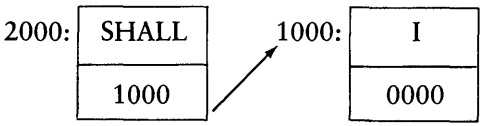
REVERSE reads the first word with the GET statement on line 27, and because the value is not a period, it allocates the first 38-byte area to hold the word. As it constructs the sentence, REVERSE places the pointer value of the sentence variable into the next field, and the input word into the word field. The most recently read word then becomes the new head of the list. After processing the word I, the list appears as shown below:

SENTENCE: 1000




REVERSE then proceeds through the loop again. This time, it reads the word SHALL and allocates the second 38-byte area. The newly allocated area becomes the new head of the list, with the resulting pointer structure:

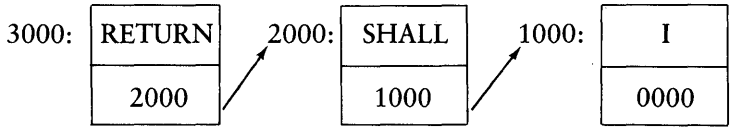
SENTENCE 2000



REVERSE repeats the loop once again and processes the last word, RETURN, and allocates the final 38-byte area, placing it at the head of the list that results in the following sequence of nodes:

SENTENCE: 3000




The program follows the pointer structure from the sentence variable to the node for RETURN, then to the node for SHALL, and finally to the node for I, where it encounters an end-of-list value 0000.

REVERSE actually builds the list in reverse order. The DO-group in the write_it procedure, lines 42 to 47, simply searches through the list, starting at the sentence pointer, and prints each word it encounters. As soon as the word is written, the FREE statement on line 46 releases the 38-byte area allocated to it. The write_it procedure moves the sentence pointer variable to the next item in the list before it executes the FREE statement to free the current element.

**Note:** storage does not remain intact after it is released.

The advantage of the list structure is that the sentence can be arbitrarily long, limited only by the size of available memory. The disadvantage, of course, is that there is considerably more storage consumed for sentences that could be represented by a 254-character string.

## 12.3 A Network Analysis Program

The next example is extensive and illustrates two points. First, it demonstrates the power of PL/I list-handling constructs. Second, it shows how to divide a large, complex program into small, logically distinct units, and thereby simplify the coding task.

The NETWORK program shown in Listing 12-4 performs a network analysis. That is, it finds the shortest path between nodes in a network. The user enters a network of cities and distances between the cities. Then NETWORK constructs a connected set of nodes using list processing structures. Upon demand from the user, NETWORK computes the shortest path from all cities in the network to the assigned destination, and then selectively displays particular optimal paths through the network.

It is easier to understand how the program operates if you first examine the console interaction shown in Listing 12-3. First, you enter a list of cities and distances between the cities, ending the entry with a CTRL-Z. Entering a CTRL-Z triggers a display of the entire network to aid in detection of input errors. NETWORK then prompts you for a destination city, in this case, Tijuana, and a starting city, in this case, Boise.

NETWORK then displays a best route. There can be several of equal length. Next, NETWORK prompts for another starting city. If you enter a CTRL-Z, NETWORK reverts to another destination prompt, leaving the network intact. Interaction continues in this manner until you enter a CTRL-Z in response to the destination prompt. When this occurs, NETWORK clears the network and returns to accept an entirely new network of cities and distances. The entire program ends if you enter an empty network at this point, for example, a CTRL-Z.

```
A>network
Type "City1,Dist, City2"
Seattle, 150, Boise
Boise, 300, Modesto
Seattle, 400, Modesto
Modesto, 150, Monterey
Modesto, 50, San-Francisco
San-Francisco, 200, Las-Vegas
Las-Vegas, 350, Monterey
Los-Angeles, 400, Las-Vegas
Bakersfield, 300, Monterey
Bakersfield, 250, Las-Vegas
Los-Angeles, 450, Tijuana
Tijuana, 700, Las-Vegas
Las-Vegas, 920, Boise
Pacific-Grove, 5, Monterey
^Z

Pacific-Grove :
        5 miles to Monterey
Tijuana :
      700 miles to Las-Vegas
      450 miles to Los-Angeles
Bakersfield :
      250 miles to Las-Vegas
      300 miles to Monterey
Los-Angeles :
      450 miles to Tijuana
      400 miles to Las-Vegas
```

**Listing 12-3. Interaction with the NETWORK Program**

```
Las-Vegas :
      920 miles to Boise
      700 miles to Tijuana
      250 miles to Bakersfield
      400 miles to Los-Angeles
      350 miles to Monterey
      200 miles to San-Francisco
San-Francisco :
      200 miles to Las-Vegas
       50 miles to Modesto
Monterey :
        5 miles to Pacific-Grove
      300 miles to Bakersfield
      350 miles to Las-Vegas
      150 miles to Modesto
Modesto :
       50 miles to San-Francisco
      150 miles to Monterey
      400 miles to Seattle
      300 miles to Boise
Boise :
      920 miles to Las-Vegas
      300 miles to Modesto
      150 miles to Seattle
Seattle :
      400 miles to Modesto
      150 miles to Boise

Type Destination Tijuana

Type Start Boise

     1250 miles remain,       300 miles to Modesto
      950 miles remain,        50 miles to San-Francisco
      900 miles remain,       200 miles to Las-Vegas
      700 miles remain,       700 miles to Tijuana
Type Start ^Z
```

Listing 12-3. (continued)

```
Type Destination Pacific-Grove

Type Start Seattle

      555 miles remain,       400 miles to Modesto
      155 miles remain,       150 miles to Monterey
        5 miles remain,         5 miles to Pacific-Grove
Type Start ^Z

Type Destination ^Z

Type "City1, Dist, City2"
^Z
A>
```

**Listing 12-3. (continued)**

### 12.3.1 NETWORK List Structures

NETWORK uses two data structures as list elements. The first structure is called a city_node and corresponds to a particular city. It is defined on line 16 of Listing 12-4. The city_node structure is shown below:

CITY_NODE:

| city_name |
|---|
| total_distance |
| investigate |
| city_list |
| route_head |

The city_name field holds the character-string value of the city's name, while the total_distance and investigate fields are used by the shortest_distance procedure. The city_list and route_head pointer values link together all the cities in the network.

The second structure is called a route_node, and is defined on line 23. A route_node establishes a single connection between a city and one of its neighbors. You allocate several route_nodes for a city, corresponding to the number of connections to its neighboring cities. The route_node structure is shown below:

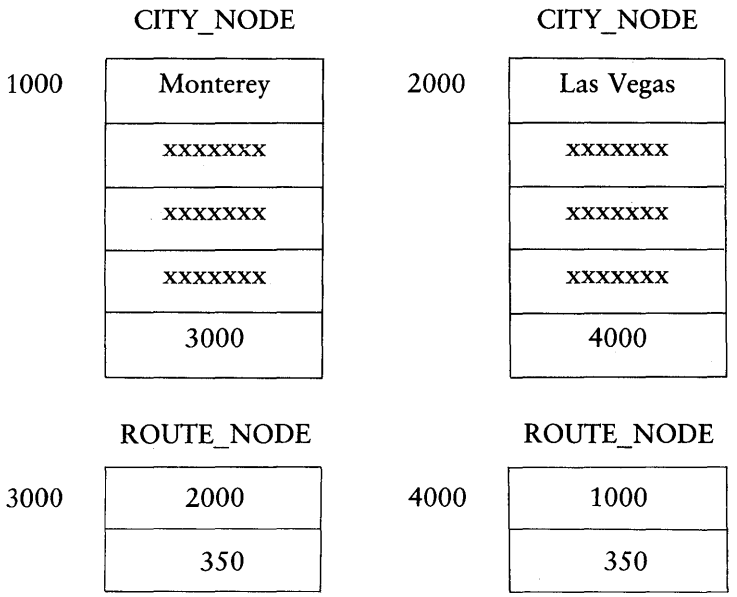
ROUTE_NODE:

| next_city |
|---|
| route_distance |
| route_list |

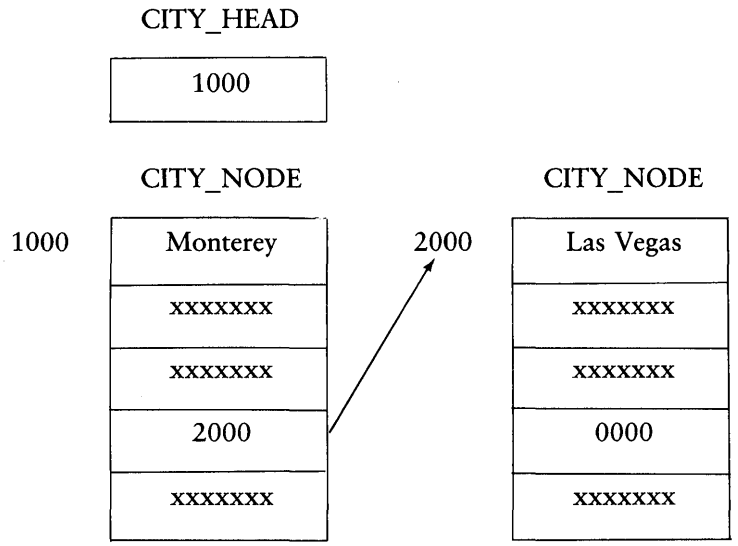
The list of route_nodes associated with a particular city begins at the pointer value called route_head that is a part of the city_node structure. The route is determined by following the route_list pointer to additional route_nodes, until you encounter a route_node with a null entry in the route_list. Each route_node also has a pointer value, denoted by next_city, that leads to a neighboring city_node, along with a route_distance field that gives the mileage to the next city.

The following example illustrates this concept. Assume Monterey is 350 miles from Las Vegas. NETWORK must allocate two city_nodes and two route_nodes with sample addresses to the left of each allocation as follows. You can temporarily ignore the fields marked x in the diagram.



A linked list, starting at city_head, leads to all cities in the network. Given the preceding two cities, the list of cities appears as follows:




### 12.3.2 Traversing the Linked Lists

Several of the procedures in NETWORK use one particular form of an iterative DO-group to traverse the linked lists. The statement on line 95 is typical:

```
do p = city_head repeat (p->city_list) while (p^=null);
```

The DO-group header successively processes each element of the linked list starting at city_head until it encounters a null link, 0000. On the first iteration, the DO-group sets the pointer variable p to the value of the pointer variable city_head. In the example above, this results in the assignment p = 1000.

On the next iteration, p takes on the value of the city_list field at 1000 that addresses Las Vegas. This results in the value p = 2000. On the last iteration, p takes on the value of the city_list field based at 2000, resulting in p = 0000. The DO-group then stops executing because p is equal to null.

### 12.3.3 Overall Program Structure

Keeping in mind the preceding discussion, look at the overall program structure. The top-level program calls occur in the DO-group between lines 31 and 38. The remainder of the program consists entirely of the nested subroutines described below.

NETWORK is logically divided into four parts:

- The input section constructs and echoes the network of cities, consisting of four procedures beginning on line 45: setup, connect, find, and print_all.
- The analysis of the shortest path between the cities takes place in the shortest_distance procedure starting on line 164.
- The shortest path display operations are split between the two procedures print_paths and print_route, respectively.
- The free_all procedure clears the old network before loading a new network.

Beginning on line 32, the main program calls setup to read the network. If the city_list is empty, then NETWORK stops. Otherwise, it calls print_all to display the network, and then calls print_paths to prompt and display the shortest routes. Upon return, NETWORK calls free_all to release storage. This process continues until you enter an empty network.

### 12.3.4 The Setup Procedure

The main loop in setup occurs between lines 54 and 58. On each iteration, the GET LIST statement, line 55, reads a pair of cities with a connecting distance. Next, setup calls the connect subroutine twice to establish the connection in both directions between the cities. The ON-unit on line 50 intercepts the CTRL-Z.

### 12.3.5 The Connect Procedure

The connect procedure establishes a single route_node to connect the first city to the second city. The connect procedure does this by calling the find procedure twice, once for the first city and once for the second city. The find procedure locates a city if it exists in the network, or creates the city_node if it does not yet exist. Upon return from find, the connect procedure creates and fills in the route_node, lines 79 to 82.

In the previous example, the first call to connect establishes the city_nodes for Monterey and Las Vegas, indirectly through the find procedure, and then produces the route_node under Monterey only. The second call to connect establishes the route_node under Las Vegas.

### 12.3.6 The Find Procedure

The find procedure, starting at line 89, searches the city_list, beginning at city_head, until it finds the input city or exhausts the city_list. If the input city does not exist, find creates it between lines 100 and 105. In any case, find returns a pointer to the requested city_node.

### 12.3.7 The Print_All Procedure

The print_all procedure appears between lines 113 and 127. NETWORK calls print_all after creating the network. This procedure starts at city_head and displays all the cities in the city_list. As it visits each city, print_all also traverses and displays the route_head. Upon completion of the print_all procedure, all city_nodes and route_nodes have been visited and displayed.

### 12.3.8 The Print_Paths Procedure

The print_paths procedure reads a destination city on line 143 and sends it to the shortest_distance procedure. Upon return, print_paths sets the total_distance field of each city_node to the total distance from the destination city. You enter the starting city on line 148, and print_paths sends it to the print_route procedure for the display operation.

### 12.3.9 The Print_Route Procedure

The print_route procedure at line 214 displays the best route from the input city to the destination. The procedure finds the best route as follows: The total distance from the input city to the destination has already been computed and stored in the total_distance field. The procedure obtains the first leg of the best route by finding a neighboring city whose total_distance field differs by exactly the distance to the neighbor. It then displays the neighbor, moves to the neighboring city, and repeats the same operation. Eventually, it reaches the destination city and completes the display operation.

Line 221 finds the original city_node. Line 231 displays the remaining distance, and the search for the first or next leg occurs between lines 233 and 244. On each iteration, line 236 tests to determine if a neighbor has been found whose total distance plus the leg distance matches the current city. If so, line 238 displays the leg distance and the search terminates by setting q to null.

### 12.3.10 The Shortest_Distance Procedure

This procedure takes an input city, called the destination, and computes the minimum total distance from every city in the network to the destination. It then records this total at each city_node in the total_distance field. In calculating the minimum total distance, the procedure implements the following algorithm:

1. Initially mark all total_distance fields with infinity, 32767 in PL/I, to indicate that the node currently has no connection.

2. Set the investigate flag to false for each city. The investigate flag marks a city_node that needs further processing.

3. Set the total_distance to the destination at zero; all others are currently set to infinity, but change during processing.

4. Set the investigate flag to true for the destination only.

5. Examine the city_list for the city_node that has the least total_distance, and whose investigate flag is true. At first, only the destination is found. When no city_node has a true investigate flag, all processing is complete and all minimum total_distance fields have been computed.

6. Clear the investigate flag for the city found in 4, and extract the current value of its total_distance field. Examine each of its neighbors; if the current total_distance field plus the leg distance is less than the total_distance field marked at the neighbor, then replace the neighbor's total_distance field by this sum. Then mark the neighbor for processing by setting its investigate flag to true. After processing each neighbor, return to step 4.

The algorithm thus proceeds through the network, developing the shortest path to any node, and as a result, visiting each city exactly once. This is because the process is linear, and any additional nodes do not significantly effect the time to analyze the network.

### 12.3.11 The Free_All Procedure

The final procedure, free_all starting at line 251, returns the network storage at the end of processing each network. The procedure visits and then discards each city_node and the entire list of route_node connections.

### 12.3.12 NETWORK Expansion

You can expand NETWORK in several ways. First, you can open a STREAM file and read the graph from disk, because it is inconvenient to type an entire network each time you run the program. You can also store several networks on disk and retrieve them on command from the console.

```
 1 a   /*****************************************************/
 2 a   /* This program finds the shortest path between nodes */
 3 a   /* in a network.  It has 8 internal procedures:       */
 4 a   /* SETUP, CONNECT, FIND, PRINT_ALL, PRINT_PATHS,      */
 5 a   /* SHORTEST_DISTANCE, PRINT_ROUTE, and FREE_ALL.      */
 6 a   /*****************************************************/
 7 a   network:
 8 b       procedure options(main);
 9 b       %replace
10 b           true     by '1'b,
11 b           false    by '0'b,
12 b           citysize by 20,
13 b           infinite by 32767;
14 b       declare
15 b           sysin file;
16 b       declare
17 b           1 city_node based,
18 b             2 city_name   character(citysize) varying,
19 b             2 total_distance  fixed,
20 b             2 investigate bit,
21 b             2 city_list   pointer,
22 b             2 route_head  pointer;
23 b       declare
24 b           1 route_node based,
25 b             2 next_city   pointer,
26 b             2 route_distance  fixed,
27 b             2 route_list  pointer;
28 b       declare
29 b           city_head pointer;
30 b
31 c     ┌─ do while(true);
32 c     │     call setup();
33 c     │     if city_head = null then
34 c     │        stop;
35 c     │     call print_all();
36 c     │     call print_paths();
37 c     │     call free_all();
38 c     └─ end;
39 b
```

**Listing 12-4. The NETWORK Program**

```
40 b   /*****************************************************/
41 b   /* This procedure reads two cities and then calls the */
42 b   /* procedure CONNECT to establish the connection (in  */
43 b   /* both directions) between the cities.               */
44 b   /*****************************************************/
45 b     setup:
46 c          procedure;
47 c          declare
48 c              distance fixed,
49 c              (city1, city2) character(citysize) varying;
50 c          on endfile(sysin) goto eof;
51 c          city_head = null;
52 c          put skip list('Type "City1, Dist, City2;"');
53 c          put skip;
54 d          do while(true);
55 d             get list(city1, distance, city2);
56 d             call connect(city1, distance, city2);
57 d             call connect(city2, distance, city1);
58 d          end;
59 c          eof:
60 c       end setup;
61 b
62 b   /*****************************************************/
63 b   /* This procedure establishes a single route_node to  */
64 b   /* connect the first city to the second city by       */
65 b   /* calling the FIND procedure twice; once for the     */
66 b   /* first city and once for the second city.           */
67 b   /*****************************************************/
68 b     connect:
69 c          procedure(source_city, distance, destination_city);
70 c          declare
71 c              source_city character(citysize) varying,
72 c              destination_city character(citysize) varying,
73 c              distance fixed,
74 c              (r, s, d) pointer;
75 c
76 c          s = find(source_city);
77 c          d = find(destination_city);
78 c          allocate route_node set (r);
79 c          r->route_distance = distance;
80 c          r->next_city  = d;
81 c          r->route_list = s->route_head;
82 c          s->route_head = r;
83 c       end connect;
84 b
```

**Listing 12-4. (continued)**

```
 85 b   /*************************************************/
 86 b   /* This procedure searches the list of cities and */
 87 b   /* returns a pointer to the requested city_node.  */
 88 b   /*************************************************/
 89 b    ┌──find:
 90 c    │      procedure(city) returns(pointer);
 91 c    │      declare
 92 c    │          city character(citysize) varying,
 93 c    │          (p, q) pointer;
 94 c    │
 95 d    │     ┌──do p = city_head
 96 d    │     │     repeat(p->city_list) while(p^=null);
 97 d    │     │     if city = p->city_name then
 98 d    │     │        return(p);
 99 d    │     └──end;
100 c    │      allocate city_node set(p);
101 c    │      p->city_name  = city;
102 c    │      p->city_list  = city_head;
103 c    │      city_head     = p;
104 c    │      p->total_distance = infinite;
105 c    │      p->route_head = null;
106 c    │      return(p);
107 c    └──end find;
108 b
109 b   /*************************************************/
110 b   /* This procedure starts at the city_head and */
111 b   /* displays all the cities in the city_list.  */
112 b   /*************************************************/
113 b    ┌──print_all:
114 c    │      procedure;
115 c    │      declare
116 c    │          (p, q) pointer;
117 c    │
118 d    │     ┌──do p = city  head
119 d    │     │     repeat(p->city_list) while(p^=null);
120 d    │     │     put skip list(p->city_name,':');
121 e    │     │    ┌──do q = p->route_head
122 e    │     │    │     repeat(q->route_list) while(q^=null);
123 e    │     │    │     put skip list(q->route_distance,'miles to',
124 e    │     │    │                   q->next_city->city_name);
125 e    │     │    └──end;
126 d    │     └──end;
127 c    └──end print_all;
128 b
```

**Listing 12-4. (continued)**

```
129 b   /*****************************************************/
130 b   /* This procedure reads a destination city, calls the */
131 b   /* SHORTEST_DISTANCE procedure, and sets the          */
132 b   /* total_distance field in each city_node to the      */
133 b   /* total distance from the destination city.          */
134 b   /*****************************************************/
135 b     print_paths:
136 c         procedure;
137 c         declare
138 c             city character(citysize) varying;
139 c
140 c         on endfile(sysin) goto eof;
141 d       do while(true);
142 d          put skip list('Type Destination ');
143 d          get list(city);
144 d          call shortest_distance(city);
145 d          on endfile(sysin) goto eol;
146 e         do while(true);
147 e            put skip list('Type Start ');
148 e            get list(city);
149 e            call print_route(city);
150 e         end;
151 d          eol: revert endfile(sysin);
152 d       end;
153 c         eof:
154 c     end print_paths;
155 b
```

**Listing 12-4. (continued)**

```
156 b   /*****************************************************/
157 b   /* This procedure is the heart of the program.  It    */
158 b   /* takes an input city, the destination, and computes */
159 b   /* the minimum total distance from every city in the  */
160 b   /* network to the destination.  It then records this  */
161 b   /* minimum value in the total_distance field of every */
162 b   /* city_node.                                         */
163 b   /*****************************************************/
164 b     shortest_distance:
165 c         procedure(city);
166 c         declare
167 c             city character(citysize) varying;
168 c         declare
169 c             bestp pointer,
170 c             (d, bestd) fixed,
171 c             (p, q, r) pointer;
172 d         do p = city_head
173 d             repeat(p->city_list) while(p^=null);
174 d             p->total_distance  = infinite;
175 d             p->investigate = false;
176 d         end;
177 c         p = find(city);
178 c         p->total_distance  = 0;
179 c         p->investigate = true;
180 d         do while(true);
181 d             bestp = null;
182 d             bestd = infinite;
183 e             do p = city_head
184 e                 repeat(p->city_list) while(p^=null);
185 e                 if p->investigate then
186 f                 do;
187 f                     if p->total_distance < bestd then
188 g                     do;
189 g                         bestd = p->total_distance;
190 g                         bestp = p;
191 g                     end;
192 f                 end;
193 e             end;
194 d             if bestp = null then
195 d                 return;
196 d             bestp->investigate = false;
```

**Listing 12-4. (continued)**

```
197 e           do q = bestp->route_head
198 e              repeat(q->route_list) while(q^=null);
199 e              r = q->next_city;
200 e              d = bestd + q->route_distance;
201 e              if d < r->total_distance then
202 f                 do;
203 f                    r->total_distance  = d;
204 f                    r->investigate = true;
205 f                 end;
206 e           end;
207 d        end;
208 c     end shortest_distance;
209 b
210 b  /*********************************************/
211 b  /* This procedure displays the best route from */
212 b  /* the input city to the destination.          */
213 b  /*********************************************/
214 b     print_route:
215 c        procedure(city);
216 c        declare
217 c            city character(citysize) varying;
218 c        declare
219 c            (p,q) pointer,
220 c            (t,d) fixed;
221 c        p = find(city);
222 d        do while(true);
223 d           t = p->total_distance;
224 d           if t = infinite then
225 e              do;
226 e                 put skip list('(No Connection)');
227 e                 return;
228 e              end;
229 d           if t = 0 then
230 d              return;
231 d           put skip list(t,'miles remain,');
232 d           q = p->route_head;
```

**Listing 12-4. (continued)**

```
233 e          do while(q^=null);
234 e             p = q->next_city;
235 e             d = q->route_distance;
236 e             if t = d + p->total_distance then
237 f                do;
238 f                   put list(d,'miles to',p->city_name);
239 f                   q = null;
240 f                end;
241 e             else
242 e                q = q->route_list;
243 e          end;
244 d       end;
245 c    end print_route;
246 b
247 b  /*************************************************/
248 b  /* This procedure frees all the storage allocated */
249 b  /* by the program while processing the network.   */
250 b  /*************************************************/
251 b    free_all:
252 c       procedure;
253 c       declare
254 c           (p, q) pointer;
255 d       do p = city_head
256 d          repeat(p->city_list) while(p^=null);
257 e          do q = p->route_head
258 e             repeat(q->route_list) while(q^=null);
259 e             free q->route_node;
260 e          end;
261 d          free p->city_node;
262 d       end;
263 c    end free all;
264 b
265 b  end network;
```

**Listing 12-4. (continued)**

*End of Section 12*

References: Sections 3.4, 7.1-7.8, 8.2 LRM

# Section 13
# Recursive Processing

Recursive processing occurs when an active procedure calls itself, or is called by another active procedure. There are many programming problems that lend themselves to this kind of construct. This section has three such problems. The first two illustrate the basic concepts, and the last one uses recursion in a practical problem.

In a recursive procedure, a CALL statement, or function reference contained in the procedure itself, reinvokes the procedure before returning to the first level call. Therefore, you must declare all such procedures with the RECURSIVE attribute so PL/I can properly save and restore the local data areas at each level of recursive call.

PL/I places two restrictions on RECURSIVE procedures. First, it passes all procedure parameters by value. You cannot return values from a recursive procedure by assignment to formal parameters. Instead, you can return a functional value or assign values to global variables.

**Note:** to maintain compatibility with full PL/I, you should not use formal parameters on the left of an assignment statement in a PL/I RECURSIVE procedure.

Second, PL/I does not allow BEGIN blocks in RECURSIVE procedures. However, it does allow nested procedures and DO-groups. The examples that follow illustrate the proper formulation of RECURSIVE procedures.



## 13.1 The Factorial Function

The classic example of recursion is evaluation of the Factorial function. This function, used throughout mathematics, is a good illustration because you can define it by iteration and recursion.

The iterative definition of the Factorial function is

$$n! = (n)(n-1)(n-2) \ldots (2)(1)$$

where n! is the Factorial function, and n is a nonnegative integer. Therefore:

$$(n-1)! = (n-1)(n-2) \ldots (2)(1)$$

You can define the Factorial function using the recursive relation:

$$n! = n(n-1)! \qquad \text{(by definition, } 0! = 1\text{)}$$

Evaluating the Factorial function using either iteration or recursion produces the following values:

```
 0! = 1
 1! = 1
 2! = (2) (1) = 2
 3! = (3) (2) (1) = 6
 4! = (4) (3) (2) (1) = 24
 5! = (5) (4) (3) (2) (1) = 120
 6! = (6) (5) (4) (3) (2) (1) = 720
 7! = (7) (6) (5) (4) (3) (2) (1) = 5040
 8! = (8) (7) (6) (5) (4) (3) (2) (1) = 40320
 9! = (9) (8) (7) (6) (5) (4) (3) (2) (1) = 362880
10! = (10) (9) (8) (7) (6) (5) (4) (3) (2) (1) = 3628800
  .
  .
  .
```

Listing 13-1 shows a program called IFACT that computes values of the Factorial function using iteration. The variable F is declared as a FIXED BINARY data item that accumulates the value of the factorial up to a maximum of 32767.

Listing 13-2 shows the output from IFACT. IFACT gives the proper value for the Factorial function up to 7!, 5040. At this point, the variable F overflows and produces improper results, but the output continues.

**Note:** PL/I does not signal FIXEDOVERFLOW for binary computations.

Listing 13-3 shows the program RFACT that performs the equivalent evaluation of the Factorial function using recursion. For comparison, RFACT uses the REPEAT form of the DO-group to control the test. RFACT declares factorial as a RECURSIVE procedure, and calls the procedure at the top level in the PUT statement on line 10.

Line 19 contains an embedded recursive call in the RETURN statement. Factorial returns when the input value is zero. All other cases require one or more recursive evaluations of factorial to produce the result. For example, 3! produces the sequence of computations,

```
factorial(3)  =  3*factorial(2)  =
                   factorial(2)     2*factorial(1)
                                      factorial(1)  =  1*factorial(0)
                                                         factorial(0)  =  1
                                                    =  1      *     1
                                 =  2        *         1      *     1
              =  3      *           2        *         1      *     1
```

producing the value 6. Listing 13-4 shows the output for the recursive factorial evaluation produced by RFACT. The values again overflow beyond 5040 due to the precision of the computations.

```
a   /****************************************/
a   /* This program evaluates the Factorial */
a   /* function (n!) using iteration.       */
a   /****************************************/
a ┌─ifact:
b │       procedure options(main);
b │       declare
b │          (i, n, F) fixed;
b │
c │    ┌──do i = 0 by 1;
c │    │     F = 1;
d │    │  ┌─do n = i to 1 by -1;
d │    │  │    F = n * F;
d │    │  └─end;
c │    │     put edit('factorial(',i,')=',F)
c │    │         (skip, a,f(2), a, f(7));
c │    └──end;
b └─end ifact;
```

**Listing 13-1. The IFACT Program**

```
A>ifact

factorial( 0)=      1
factorial( 1)=      1
factorial( 2)=      2
factorial( 3)=      6
factorial( 4)=     24
factorial( 5)=    120
factorial( 6)=    720
factorial( 7)=   5040
factorial( 8)= -25216   * the values are incorrect
factorial( 9)= -30336   from this point on
factorial(10)=  24320
factorial(11)=   5376
factorial(12)=  -1024
factorial(13)= -13312
factorial(14)=  10240
factorial(15)=  22528
factorial(16)= -32768
factorial(17)= -32768
factorial(18)=      0
factorial(19)=      0
     .
     .
     .
```

**Listing 13-2. Output from the IFACT Program**

```
 1 a   /*****************************************/
 2 a   /* This program evaluates the Factorial */
 3 a   /* function (n!) using recursion.       */
 4 a   /*****************************************/
 5 a  rfact:
 6 b        procedure options(main);
 7 b        declare
 8 b           i fixed;
 9 c        do i = 0 repeat(i+1);
10 c           put skip list('factorial(',i,')=',factorial(i));
11 c        end;
12 b        stop;
13 b
```

**Listing 13-3. The RFACT Program**

```
14 b     ┌─factorial:
15 c     │      procedure(i) returns(fixed) recursive;
16 c     │      declare
17 c     │         i fixed;
18 c     │      if i = 0 then return (1);
19 c     │      return (i * factorial(i-1));
20 c     └─end factorial;
21 b
22 b ─end rfact;
```

**Listing 13-3. (continued)**

```
A>fact

factorial(           0)=      1
factorial(           1)=      1
factorial(           2)=      2
factorial(           3)=      6
factorial(           4)=     24
factorial(           5)=    120
factorial(           6)=    720
factorial(           7)=   5040
factorial(           8)= -25216   * the values are incorrect
factorial(           9)= -30336   from this point on
factorial(          10)=  24320
factorial(          11)=   5376
factorial(          12)=  -1024
factorial(          13)= -13312
factorial(          14)=  10240
factorial(          15)=  22528
factorial(          16)= -32768
factorial(          17)= -32768
factorial(          18)=      0
factorial(          19)=      0
      .
      .
      .
```

**Listing 13-4. Output from the RFACT Program**

## 13.2 FIXED DECIMAL and FLOAT BINARY Evaluation

The Factorial evaluation programs here illustrate an important point about arithmetic calculations using different data types. Listing 13-5 shows a program called DFACT. It is the same recursive evaluation of the Factorial function found in RFACT, but it uses the FIXED DECIMAL data with the maximum allowable precision. Listing 13-6 shows the output from DFACT. The largest value produced by the program is:

Factorial(17) = 355,687,428,096,000

At this point, the run-time system signals FIXEDOVERFLOW to indicate that the decimal computation has overflowed the maximum 15 digit value.

Listing 13-7 shows the program FFACT that evaluates the Factorial function using FLOAT BINARY data. Listing 13-8 shows the output from FFACT. FFACT can compute the value of the function beyond 17. PL/I truncates the number of significant digits on the right to approximately 7 equivalent decimal digits. Again, FFACT ends when the run-time system signals the OVERFLOW condition because the program produces an exponent value that cannot be maintained in the floating-point representation.

```
 1 a   /************************************************/
 2 a   /* This program evaluates the Factorial function */
 3 a   /* (n!) using recursion and FIXED DECIMAL data.  */
 4 a   /************************************************/
 5 a  dfact:
 6 b       procedure options(main);
 7 b       declare
 8 b          i fixed;
 9 c       do i = 0 repeat(i+1);
10 c          put skip list('Factorial(',i,')=',factorial(i));
11 c       end;
12 b       stop;
13 b
14 b       factorial:
15 c            procedure(i) returns(fixed decimal(15,0))
16 c                         recursive;
17 c            declare
18 c               i fixed;
19 c
20 c            if i = 0 then return (1);
21 c            return (decimal(i,15) * factorial(i-1));
22 c       end factorial;
23 b
24 b  end dfact;
```

**Listing 13-5. The DFACT Program**

```
A>dfact

Factorial(           0)=                   1
Factorial(           1)=                   1
Factorial(           2)=                   2
Factorial(           3)=                   6
Factorial(           4)=                  24
Factorial(           5)=                 120
Factorial(           6)=                 720
Factorial(           7)=                5040
Factorial(           8)=               40320
Factorial(           9)=              362880
Factorial(          10)=             3628800
Factorial(          11)=            39916800
Factorial(          12)=           479001600
Factorial(          13)=          6227020800
Factorial(          14)=         87178291200
Factorial(          15)=       1307674368000
Factorial(          16)=      20922789888000
Factorial(          17)=     355687428096000
Factorial(          18)=
FIXED OVERFLOW (1)
Traceback: 0007 019F 0018 0000 # 2809 6874 0355 0141
A>
```

**Listing 13-6. Output from the DFACT Program**

```
 1 a   /************************************************/
 2 a   /* This program evaluates the Factorial function */
 3 a   /* (n!) using recursion and FLOAT BINARY data.   */
 4 a   /************************************************/
 5 a  ffact:
 6 b        procedure options(main);
 7 b        declare
 8 b           i fixed;
 9 c        do i = 0 repeat(i+1);
10 c           put skip list('Factorial(',i,')=',factorial(i));
11 c        end;
12 b        stop;
13 b
14 b        factorial:
15 c             procedure(i) returns(float) recursive;
16 c             declare
17 c                i fixed;
18 c             if i = 0 then return (1);
19 c             return (i * factorial(i-1));
20 c        end factorial;
21 b
22 b  end ffact;
```

**Listing 13-7. The FFACT Program**

```
A>ffact

Factorial(         0 )=  1.000000E+00
Factorial(         1 )=  1.000000E+00
Factorial(         2 )=  2.000000E+00
Factorial(         3 )=  0.600000E+01
Factorial(         4 )=  2.400000E+01
Factorial(         5 )=  1.200000E+02
Factorial(         6 )=  0.720000E+03
Factorial(         7 )=  0.504000E+04
Factorial(         8 )=  4.032000E+04
Factorial(         9 )=  3.628799E+05
Factorial(        10 )=  3.628799E+06
Factorial(        11 )=  3.991679E+07
Factorial(        12 )=  4.790015E+08
Factorial(        13 )=  0.622702E+10
Factorial(        14 )=  0.871782E+11
Factorial(        15 )=  1.307674E+12
Factorial(        16 )=  2.092278E+13
Factorial(        17 )=  3.556874E+14
Factorial(        18 )=  0.640237E+16
Factorial(        19 )=  1.216450E+17
Factorial(        20 )=  2.432901E+18
Factorial(        21 )=  0.510909E+20
Factorial(        22 )=  1.124000E+21
Factorial(        23 )=  2.585201E+22
Factorial(        24 )=  0.620448E+24
Factorial(        25 )=  1.551121E+25
Factorial(        26 )=  4.032914E+26
Factorial(        27 )=  1.088887E+28
Factorial(        28 )=  3.048883E+29
Factorial(        29 )=  0.884176E+31
Factorial(        30 )=  2.652528E+32
Factorial(        31 )=  0.822283E+34
Factorial(        32 )=  2.631308E+35
Factorial(        33 )=  0.868331E+37
Factorial(        34 )=
OVERFLOW (1)
Traceback: 006C 13CB 019B 0000 # 8608 0B15 FB51 0141
A>
```

**Listing 13-8. Output from the FFACT Program**

## 13.3 The Ackermann Function

The PL/I run-time system maintains a 512-byte stack area to hold subroutine return addresses and some temporary results. Under normal circumstances, this stack area is sufficiently large for nonrecursive and most simple recursive procedure processing. The program in this section, however, illustrates multiple recursion using a stack depth that can exceed the 512-byte default value.

The Ackermann function, denoted by A(m,n), comes from Number Theory and has the following recursive definition:

$$A(m,n) = \begin{cases} n+1 & \text{if } m=0, \text{ otherwise} \\ A(m-1,1) & \text{if } n=0, \text{ otherwise} \\ A((m-1), A(m,n-1)) & \end{cases}$$

Listing 13-9 shows the ACK program that reads two values for the maximum m and, n on line 11, and then evaluates the function for these values. Listing 13-10 shows the program interaction. Although the Ackermann function returns a FIXED BINARY value, the program uses the built-in DECIMAL function to control the size of the converted field in the PUT statements on lines 12, 15, and 17.

In this example, ACK uses the STACK option on line 7 to increase the size of the run-time stack from its default value, 512 bytes, to 2000 bytes.

**Note:** the STACK option is only valid with the MAIN option. You must determine the value of the STACK option empirically, because the Compiler cannot compute the depth of recursion. If the allocated stack size is too small and the stack overflows during recursion, the run-time system outputs the message:

```
FREE SPACE OVERWRITE
```

and then ends the program.

This kind of multiple recursion processing is CPU intensive. You should experiment with some different values for max, and see if you can determine how much stack is being used.

```
 1 a   /*************************************************/
 2 a   /* This program evaluates the Ackermann function   */
 3 a   /* A(m,n), and increases the size of the stack     */
 4 a   /* because of the large number of recursive calls. */
 5 a   /*************************************************/
 6 a  ack:
 7 b       procedure options(main,stack(2000));
 8 b       declare
 9 b           (m,maxm,n,maxn) fixed;
10 b       put skip list('Type max m,n: ');
11 b       get list(maxm,maxn);
12 b       put skip
13 b           list('         ',(decimal(n,4) do n=0 to maxn));
14 c      do m = 0 to maxm;
15 c         put skip list(decimal(m,4),':');
16 d         do n = 0 to maxn;
17 d            put list(decimal(ackermann(m,n),4));
18 d         end;
19 c      end;
20 b       stop;
21 b
22 b      ackermann:
23 c                procedure(m,n) returns(fixed) recursive;
24 c                declare (m,n) fixed;
25 c                if m = 0 then
26 c                   return(n+1);
27 c                if n = 0 then
28 c                   return(ackermann(m-1,1));
29 c                return(ackermann(m-1,ackermann(m,n-1)));
30 c      end ackermann;
31 b
32 b  end ack;
```

**Listing 13-9. The ACK Program**

```
A>ack

Type max m,n:  *3,5

              0       1       2       3       4       5
     0:       1       2       3       4       5       6
     1:       2       3       4       5       6       7
     2:       3       5       7       9      11      13
     3:       5      13      29      61     125     253
A>
```

**Listing 13-10. Interaction with the ACK Program**

## 13.4 An Arithmetic Expression Evaluator

One of the practical uses of recursion is the translation of statements in a high-level programming language. This is because most languages are defined recursively. In block-structured languages like PL/I for example, DO-groups and BEGIN and PROCEDURE blocks can all be nested, and the resulting structure lends itself easily to recursive processing.

The next example illustrates how you can use recursion to evaluate arithmetic expressions. Here is a simple, recursive definition of an arithmetic expression: An expression is a simple number, or a pair of expressions separated by a +, −, *, or /, and enclosed in parentheses.

Using this definition, the number 3.6 is an expression because it is a simple number. Clearly,

```
(3.6 + 6.4)
```

is an expression because it consists of a pair of expressions that are both simple numbers, separated by a +, and enclosed in parentheses. Also,

```
(1.2 * (3.6 + 6.4))
```

is a valid expression because it contains the two valid expressions 1.2 and (3.6 + 6.4), separated by a * and enclosed in parentheses.

Using the definition given above, the sequences,

```
3.6 + 6.4
```

```
(1.2 + 3.6 + 6.4)
```

are not valid expressions because the first is not enclosed in parentheses, while the second is not a pair of expressions in parentheses.

The preceding definition of an expression is somewhat restricted, but once established, you can expand it to include more complex expressions.

Listing 13-11 shows an expression evaluation program called EXPR1. The main processing takes place between lines 27 and 31 where EXPR1 reads an expression from the console and types the evaluated result back to you. Listing 13-12 shows the console interaction with EXPR1 where the user enters several properly and improperly formed expressions.

```
 1 a   /*******************************************************/
 2 a   /* This program evaluates an arithmetic expression     */
 3 a   /* using recursion.  It contains two procedures, GNT   */
 4 a   /* obtains the input expression consisting of separate */
 5 a   /* tokens, and EXP that performs the recursive         */
 6 a   /* evaluation of the tokens in the input line.         */
 7 a   /*******************************************************/
 8 a expression:
 9 b      procedure options(main);
10 b      declare
11 b         sysin file,
12 b         value float,
13 b         token character(10) varying;
14 b
15 b      on endfile(sysin)
16 b         stop;
17 b
18 b      on error(1) /* conversion or signal */
19 c      begin;
20 c         put skip list('Invalid Input at ',token);
21 c         get skip;
22 c         goto restart;
23 c      end;
24 b
25 b      restart:
26 b
27 c        do while('1'b);
28 c           put skip(3) list('Type expression: ');
29 c           value = exp();
30 c           put skip list('Value is:',value);
31 c        end;
32 b
33 b    gnt:
34 c       procedure;
35 c       get list(token);
36 c    end gnt;
37 b
```

**Listing 13-11. The EXPRESSION Program using Evaluator EXPR1**

```
38 b     exp:
39 c        procedure returns(float binary) recursive;
40 c        declare x float binary;
41 c        call gnt();
42 c        if token = '(' then
43 d           do;
44 d              x  = exp();
45 d              call gnt();
46 d              if token = '+' then
47 d                 x = x + exp();
48 d              else
49 d              if token = '-' then
50 d                 x = x - exp();
51 d              else
52 d              if token = '*' then
53 d                 x = x * exp();
54 d              else
55 d              if token = '/' then
56 d                 x = x / exp();
57 d              else
58 d              signal error(1);
59 d              call gnt();
60 d              if token ^= ')' then
61 d                 signal error(1);
62 d           end;
63 c        else
64 c              x = token;
65 c        return(x);
66 c     end exp;
67 b
68 b  end expression;
```

**Listing 13-11. (continued)**

### 13.4.1 The Exp Procedure

The heart of the expression analyzer is the RECURSIVE procedure exp. This procedure implements the recursive definition given above and decomposes the input expressions piece by piece. The GNT, Get Next Token, procedure reads the next element or token, a left or right parenthesis, a number, or one of the arithmetic operators in the input line. GNT uses a GET LIST statement, so you must separate each token with a blank or end-of-line character.

On line 41, exp calls GNT. GNT places the next input token into the CHARACTER(10) variable called token. If the first item is a number, then the series of tests in exp sends control to line 64. The assignment to x automatically converts the value of token to a floating-point value. Then exp returns this converted value to line 29, where EXPR1 stores it into value, and subsequently writes it out as the result of the expression.

If the expression is nontrivial, then exp scans the leading left parenthesis on line 42, and enters the DO-group on line 43. EXPR1 immediately evaluates the first subexpression no matter how complicated, and stores it into the variable x on line 44. EXPR1 then checks token for an occurrence of +, −, *, or /. Suppose, for example, token contains the * operator. The statement on line 53 recursively invokes the exp procedure to evaluate the right side of the expression. Upon return, it multiplies this result by the value of the left side that was previously computed. EXPR1 then checks the balancing, right parenthesis starting on line 60, and returns the resulting product as the value of exp on line 64.

### 13.4.2 Condition Processing

EXPR1 performs condition processing in three places. The first ON-unit, line 15, intercepts an end-of-file, CTRL-Z, condition on the input file, and executes a STOP statement. The second ON-unit, line 18, receives control if an error occurs during conversion from character to floating-point representation at the assignment on line 64. The ON-unit displays the token in error, and then executes a GET SKIP statement to clear the data to the end of the line. It then transfers control to the restart label, which prompts for another input expression.

EXPR1 signals a condition when it encounters an invalid operator or an unbalanced expression. If the operator is not a +, −, *, or /, then EXPR1 executes line 58 and signals the ON-unit, line 18. Again, the ON-unit displays the error and transfers control to the restart label. Similarly, a missing right parenthesis on line 60 triggers the ERROR(1) ON-unit to report the error and restart the program. When the program restarts, PL/I discards the information on the current level of recursion.

```
A>expr1

Type expression:  ( 4 + 5.2 )
Value is:  0.920000E+01

Type expression:  4.5e-1
Value is:  4.499999E-01

Type expression:  ( 4 & 5 )
Invalid input at  &

Type expression:  ( ( 3 + 4 ) - (10 / 8 ) )
Value is:  0.575000E+01

Type expression:  ( 3 * 4 )
Value is:  1.200000E+01

Type Expression:  ^Z
A>
```

**Listing 13-12. Interaction with EXPR1**

### 13.4.3 Improvements

The expression analyzer requires spaces between tokens in the input line. Recall that Section 11.2 contains a more advanced version of GNT.

We incorporate this expanded version of GNT into the expression analyzer, and also change the error recovery mechanism so that now line 27 discards the remainder of the current input when restarting the program. Listing 13-13 shows the improved version called EXPR2, and Listing 13-14 shows the console interaction with this improved expression evaluator.

Even in EXPR2 there is room for expansion. First, you can add more operators to expand upon the basic arithmetic functions. Also, you can add operator precedence and eliminate the requirement for explicit parentheses. Beyond that, you can add variable names and assignment statements to turn the program into a BASIC interpreter!

```
 1 a   /*****************************************************/
 2 a   /* This program evaluates an arithmetic expression   */
 3 a   /* using recursion.  It contains an expanded version */
 4 a   /* of the GNT procedure that obtains an expression   */
 5 a   /* containing separate tokens. EXP then recursively  */
 6 a   /* evaluates the tokens in the input line.           */
 7 a   /*****************************************************/
 8 a
 9 a   expression:
10 b       procedure options(main);
11 b
12 b       %replace
13 b           true by '1'b;
14 b
15 b       declare
16 b          sysin file,
17 b          value float,
18 b          (token character(10), line character(80)) varying
19 b          static initial('');
20 b
21 b       on endfile(sysin)
22 b          stop;
23 b
24 b       on error(1) /* conversion or signal */
25 c       begin;
26 c          put skip list('Invalid Input at ',token);
27 c          token = ''; line = '';
28 c          goto restart;
29 c       end;
30 b
31 b       restart:
32 b
33 c       do while('1'b);
34 c          put skip(3) list('Type expression: ');
35 c          value = exp();
36 c          put edit('Value is: ',value) (skip,a,f(10,4));
37 c       end;
38 b
```

**Listing 13-13. (continued)**

```
39 b     gnt:
40 c       procedure;
41 c       declare
42 c          i fixed;
43 c
44 c        line = substr(line,length(token)+1);
45 d       do while(true);
46 d          if line = '' then
47 d             get edit(line) (a);
48 d             i = verify(line,' ');
49 d          if i = 0 then
50 d             line = '';
51 d          else
52 e             do;
53 e                line = substr(line,i);
54 e                i = verify(line,'0123456789.');
55 e             if i = 0 then
56 e                token = line;
57 e             else
58 e             if i = 1 then
59 e                token = substr(line,1,1);
60 e             else
61 e                token = substr(line,1,i-1);
62 e             return;
63 e             end;
64 d       end;
65 c     end gnt;
66 b
```

**Listing 13-13. (continued)**

```
67 b     exp:
68 c          procedure returns(float binary) recursive;
69 c          declare x float binary;
70 c          call gnt();
71 c          if token = '(' then
72 d               do;
73 d                  x  = exp();
74 d                  call gnt();
75 d                  if token = '+' then
76 d                     x = x + exp();
77 d                  else
78 d                  if token = '-' then
79 d                     x = x - exp();
80 d                  else
81 d                  if token = '*' then
82 d                     x = x * exp();
83 d                  else
84 d                  if token = '/' then
85 d                     x = x / exp();
86 d                  else
87 d                  signal error(1);
88 d                  call gnt();
89 d                  if token ^= ')' then
90 d                     signal error(1);
91 d               end;
92 c          else
93 c               x = token;
94 c          return(x);
95 c     end exp;
96 b
97 b end expression;
```

**Listing 13-13. Expression Evaluator EXPR2**

```
A>expr2

Type expression:  ( 2 * 14.5 )

Value is:    29.0000

Type expression:  ( (2*3) / (4.3-1.5))

Value is:     2.1429

Type expression:  zot

Invalid Input at  z

Type expression:  ((2*3) -5)

Value is:     1.0000

Type expression:  (2 n5)

Invalid Input at n

Type expression:  ^Z
A>
```

**Listing 13-14. (continued)**

*End of Section 13*

References: Sections 2.8-2.9, 3.1-3.2, 4.2, 9 LRM

# Section 14
# Separate Compilation

All of the programs presented so far are single, complete units, although many contain one or more internal procedures. It is often useful to break larger programs into distinct modules to be subsequently linked with one another and with the PL/I Run-time Subroutine Library (RSL).

There are two reasons for separately compiling and linking programs in this manner. First, large programs take longer to compile. Smaller segments can be independently developed, tested, and integrated, requiring less overall compilation time for the entire project. A large program can also overrun the memory space available for the Symbol Table.

Second, particular subroutines are useful for your own application programming. You can build your own library of subroutines and selectively link them to your programs. Having such a library of common subroutines greatly reduces the overall development time for any particular program.

This section presents basic information required to link program segments. It also presents an example of a program that is compiled as two separate modules and then linked together.



## 14.1 Data and Program Declarations

You can direct separate modules to share data areas by including the EXTERNAL attribute in the declaration of the data item. For example, the statement,

```
declare x(10) fixed binary external;
```

defines a variable named x occupying 10 FIXED BINARY locations, 20 contiguous bytes, that is accessible by any other module that uses the same declaration.

Similarly, the statement,

```
declare
  1 s external,
    2 y(10) bit(8),
    2 z character(9) varying;
```

defines a structure named s, occupying a 20-byte area that is accessible by any other modules that use the same declaration.

The following list summarizes basic rules that apply to the declaration of external data:

- EXTERNAL data items are accessible in any block in which you declare them. The EXTERNAL attribute overrides the scope rules for internal data.
- Initialize an EXTERNAL data item in only one module. Other modules can then reference the item.
- Declare all EXTERNAL data areas with the same length in all modules in which they appear.
- In the 8080 implementation, EXTERNAL data items must be unique in the first six characters because the linkage editing format truncates from the seventh character on. In the 8086 implementation, there is no such restriction.
- Avoid using ? symbols in variable names, because this character is used as a prefix for names in the RSL.
- Remember that PL/I automatically assigns the STATIC attribute to any EXTERNAL data item.

## 14.2 ENTRY Data

ENTRY constants and ENTRY variables are data items that identify procedure names and describe their parameter values. ENTRY constants correspond to external procedures, or procedures defined in the main procedure.

ENTRY variables take on ENTRY constant values during program execution through a direct assignment statement, or an actual-to-formal parameter assignment implicit in a subroutine call.

You invoke a procedure directly through a call to an ENTRY constant, or indirectly by calling a procedure constant value held by an ENTRY variable. As with label variables, you can also subscript ENTRY variables.

The program shown in Listing 14-1 illustrates ENTRY data. The ENTRY variable f declared on line 8 is an array containing three ENTRY constants. Starting on line 12, the program initializes the subscripted elements to the constants a, b, and c respectively. Note that the constant a corresponds to an externally compiled procedure (see Listing 3-1a).

The ENTRY variable called f declared on line 8 contains three elements. Starting on line 13, the program initializes the individually subscripted elements to the constants sin, g, and h, respectively.

On line 16, the DO-group prompts for input of a value to send to each function, and then on line 19 calls each function once with the invocation,

```
f(i)(x)
```

where the first parenthesis pair defines the subscript, and the second encloses the list of actual arguments.

The declaration of ENTRY constants and ENTRY variables is similar to FILE constants and FILE variables. PL/I assumes all formal parameters declared as type ENTRY to be entry variables. In all other cases, an entry is constant unless you declare it with the VARIABLE keyword.

The following rules apply to external procedure declarations:

- You can access data items with the EXTERNAL attribute in any procedure where they are declared EXTERNAL.
- In the 8080 implementation, you must make external procedure names unique in the first six characters (see Section 14.1). In the 8086 implementation, there is no restriction.
- Avoid using the ? symbol in procedure names.

**Note:** in addition, you must ensure that each formal parameter exactly matches the actual procedure declaration, and that the RETURNS attribute exactly matches the form returned for function procedures.

```
 1 a   /************************************************/
 2 a   /* This program illustrates ENTRY variables and */
 3 a   /* constants.                                   */
 4 a   /************************************************/
 5 a  call:
 6 b       procedure options(main);
 7 b       declare
 8 b           f(3) entry(float) returns(float) variable,
 9 b           a entry(float) returns(float),
10 b           i fixed, x float;
11 b
12 b       f(1) = a;
13 b       f(2) = b;
14 b       f(3) = c;
15 b
16 c      do i = 1 to 3;
17 c         put skip list('Type x ');
18 c         get list(x);
19 c         put list('f(',i,')=',f(i)(x));
20 c      end;
21 b       stop;
22 b
23 b      b:
24 c        procedure(x) returns(float); /* internal procedure */
25 c        declare x float;
26 c        return(2*x + 1);
27 c      end b;
28 b
29 b      c:
30 c        procedure(x) returns(float); /* internal procedure */
31 c        declare x float;
32 c        return(sin(x));
33 c      end c;
34 b
35 b  end call;
```

**Listing 14-1. An Illustration of ENTRY Constants and Variables**

## 14.3 An Example of Separate Compilation

This section presents an example program of two modules that are compiled separately and then linked together. The two modules are called MAININVT and INVERT, and are shown in Listings 14-2 and 14-3, respectively. Compiling each of these modules and then linking them together produces a program that interacts with the console to produce the solution set for a system of simultaneous equations.

To understand how the programs work, first consider the following system of equations in three unknowns:

```
 a - b + c = 2        a - b + c = 3.5
 a + b - c = 0        a + b - c = 1
2a - b     = 0       2a - b     = -1
```

The values,

```
a = 1          a = 2.25
b = 2   and    b = 5.50
c = 3          c = 6.75
```

constitute valid solutions to this system of equations, because:

```
  1 - 2 + 3 = 2              2.25 - 5.50 + 6.75 = 3.50
  1 + 2 - 3 = 0   and        2.25 + 5.50 - 6.75 = 0
2*1 - 2     = 0            2*2.25 - 5.50      = 0
```

The values 2,0,0 and 3.5,0,0 are called solution vectors for the matrix. The coefficients of the matrix are:

```
1  -1   1
1   1  -1
2  -1   0
```

The MAININVT module interacts with the console to read the coefficients and the solution vectors for a system of equations. The INVERT module performs the actual matrix inversion that solves the system of equations.

The essential difference between these two modules is found in the procedure heading. The maininvt procedure is the main program because it is defined with the MAIN option. The invert procedure is a subroutine called by the main program. In Listing 14-2, the declaration starting on line 15 defines invert as an EXTERNAL entry constant that is then called on line 49.

On line 21, MAININVT declares the parameters for the invert procedure as a matrix of floating-point numbers denoted by maxrow and maxcol. Invert is defined with two additional FIXED(6) parameters, but does not return a value.

The invert procedure, shown in Listing 14-3 has three formal parameters called a, r, and c. They are defined on line 2 and declared in lines 7 and 8. INVERT takes the actual values of maxrow and maxcol, corresponding to the largest possible row and

column value, from a %INCLUDE file, as indicated by the + symbols following the line number at the left of both listings.

After you compile both of the modules, link them together with the command:

```
A>link invmat=maininvt,invert
```

The linkage editor combines the two modules, selects the necessary subroutines from the RSL, and creates the command file, named INVMAT.

Listing 14-4 shows the interaction with INVMAT. In this sample interaction, the user first enters the identity matrix to test the basic operations. The inverse matrix produced for this input value is also the identity matrix.

The user then enters the system of equations shown above, along with two solution vectors. The output values for this system are shown under Solutions: and match the values shown above. The second set of solutions corresponds to the second solution vector input.

Finally, the user tests INVMAT with an invalid input matrix size, and then ends the program by entering a zero row size.

```
 1 a      /*****************************************************/
 2 a      /* This program is the main module in a program that */
 3 a      /* performs matrix inversion.  It calls the entry    */
 4 a      /* constant INVERT which does the actual inversion.  */
 5 a      /*****************************************************/
 6 a    maininvt:
 7 b        procedure options(main);
 8 b        %replace
 9 b            true   by '1'b,
10 b            false  by '0'b;
11+b            %replace
12+b            maxrow by 26,
13+b            maxcol by 40;
14 b
15 b       declare
16 b            mat(maxrow,maxcol) float binary(24),
17 b            (i,j,n,m) fixed(6),
18 b            var character (26) static initial
19 b            ('abcdefghijklmnopqrstuvwxyz'),
20 b            invert entry
21 b            ((maxrow,maxcol) float(24), fixed(6), fixed(6));
22 b
23 b       put list('Solution of Simultaneous Equations');
```

**Listing 14-2. MAININVT - Matrix Inversion Main Program Module**

```
24 c        do while(true);
25 c           put skip(2) list('Type rows, columns: ');
26 c           get list(n);
27 c           if n = 0 then
28 c              stop;
29 c
30 c           get list(m);
31 c           if n > maxrow ! m > maxcol then
32 c              put skip list('Matrix is Too Large');
33 c           else
34 d              do;
35 d                 put skip list('Type Matrix of Coefficients');
36 d                 put skip;
37 e                 do i = 1 to n;
38 e                    put list('Row',i,':');
39 e                    get list((mat(i,j) do j = 1 to n));
40 e                 end;
41 d
42 d                 put skip list('Type Solution Vectors');
43 d                 put skip;
44 e                 do j = n + 1 to m;
45 e                    put list('Variable',substr(var,j-n,1),':');
46 e                    get list((mat(i,j) do i = 1 to n));
47 e                 end;
48 d
49 d                 call invert(mat,n,m);
50 d                 put skip(2) list('Solutions:');
51 e                 do i = 1 to n;
52 e                    put skip list(substr(var,i,1),'=');
53 e                    put edit((mat(i,j) do j = 1 to m-n))
54 e                           (f(8,2));
55 e                 end;
56 d
57 d                 put skip(2) list('Inverse Matrix is');
58 e                 do i = 1 to n;
59 e                    put skip edit((mat(i,j) do j = m-n+1 to m))
60 e                              (x(3),6f(8,2),skip);
61 e                 end;
62 d              end;
63 c        end;
64 b
65 b     end mainInvt;
```

**Listing 14-2. (continued)**

```
 1 a ┌──invert:
 2 b │       procedure (a,r,c);
 3+b │           %replace
 4+b │           maxrow by 26,
 5+b │           maxcol by 40;
 6 b │       declare
 7 b │           (d, a(maxrow,maxcol)) float binary(24),
 8 b │           (i,j,k,l,r,c) fixed binary(6);
 9 c │     ┌──do i = 1 to r;
10 c │     │     d = a(i,1);
11 d │     │   ┌──do j = 1 to c - 1;
12 d │     │   │     a(i,j) = a(i,j+1)/d;
13 d │     │   └──end;
14 c │     │     a(i,c) = 1/d;
15 d │     │   ┌──do k = 1 to r;
16 d │     │   │     if k ^= i then
17 e │     │   │   ┌──do;
18 e │     │   │   │     d = a(k,1);
19 f │     │   │   │   ┌──do l = 1 to c - 1;
20 f │     │   │   │   │     a(k,l) = a(k,l+1) - a(i,l) * d;
21 f │     │   │   │   └──end;
22 e │     │   │   │     a(k,c) = - a(i,c) * d;
23 e │     │   │   └──end;
24 d │     │   └──end;
25 c │     └──end;
26 b │
27 b └──end invert;
```

**Listing 14-3. INVERT - Matrix Inversion Subroutine**

```
A>invmat
Solution of Simultaneous Equations

Type rows, columns: 3,3

Type Matrix of Coefficients
Row      1 :1 0 0
Row      2 :0 1 0
Row      3 :0 0 1

Type Solution Vectors
```

**Listing 14-4. Interaction with the INVMAT Program**

```
Solutions:
a=
b=
c=

Inverse Matrix is
       1.00     0.00     0.00
       0.00     1.00     0.00
       0.00     0.00     1.00

Type rows, columns: 3,5

Type Matrix of Coefficients
Row      1 :1 -1 1
Row      2 :1 1 -1
Row      3 :2 -1 0

Type Solution Vectors
Variable a :2 0 0
Variable b :3.5 1 -1


Solutions:
a =     1.00     2.25
b =     2.00     5.50
c =     3.00     6.75

Inverse Matrix is
       0.50     0.50     0.00
       1.00     1.00    -1.00
       1.50     0.50    -1.00

Type rows, columns: 41,0

Matrix is Too Large

Type rows, columns: 0

A>
```

**Listing 14-4. (continued)**

*End of Section 14*

References: Sections 3.3.2, 5.1-5.4, 8.2 LRM

# Section 15
# Decimal Computations

This section explains how PL/I handles decimal computations, stores decimal data, and converts data types. Study this material thoroughly because it is vital to understanding commercial processing.

## 15.1 A Comparison of Decimal and Binary Operations

The arithmetic with which we are most familiar uses the decimal number system. All operations, such as addition and multiplication, are based on the number ten, and involve the digits zero through nine. Computers, however, perform arithmetic operations using binary or base 2 numbers. Computers use binary numbers because the 1s and 0s can be directly processed by the on-off electronic switches found in arithmetic processors.

Most programming languages allow you to write programs that process base 10 constants and data items in simple and readable forms. Because the programs process decimal values, it is necessary to convert values into a binary form on input and back to a decimal form on output. This conversion from one type to another can introduce truncation errors that are unacceptable in commercial processing. Thus, decimal arithmetic is often required to avoid propagating errors throughout computations.

In most programming languages, you have no control over the internal format used for numeric processing. In fact, two of the most popular BASIC interpreters for microprocessors differ primarily in the internal number formats. One uses floating-point binary, while the other performs calculations using decimal arithmetic.

PASCAL Compilers generally use floating- and fixed-point binary formats with implementation-defined precision, while FORTRAN Compilers always use floating- or fixed-point binary.

COBOL on the other hand, was designed for use in commercial applications where exact dollars and cents must be maintained throughout computations. Therefore, COBOL processes data items using decimal arithmetic.



PL/I gives you a choice between representations, so that you can tailor the data in each program to the exact needs of the particular application. PL/I uses FIXED DECIMAL data items to perform commercial functions, and FLOAT BINARY items for scientific processing where computation speed is the most important factor, and truncation errors are insignificant or ignored altogether.

The two programs shown below illustrate the essential difference between the two data types.

```
decimal_comp:
    procedure options(main);
    declare
      i fixed,
      t fixed decimal(7,2);
    t = 0;
    do i = 1 to 10000;
       t = t + 3.10;
    end;
    put edit(t) (f(10,2));
end decimal_comp;
```

```
binary_comp:
    procedure options(main);
    declare
      i fixed,
      t float binary(24);
    t = 0;
    do i = 1 to 10000;
       t = t + 3.10;
    end;
    put edit(t) (f(10,2));
end binary_comp;
```

Both of these programs sum the value 3.10 a total of 10,000 times. The only difference between these programs is that DECIMAL_COMP computes the result using a FIXED DECIMAL variable, while BINARY_COMP performs the computation using FLOAT BINARY.

DECIMAL_COMP produces the correct result 31000.00, while BINARY_COMP produces the approximation 30997.30. The 2.70 difference is due to the inherent truncation errors that occur when PL/I converts certain decimal constants, such as 3.10, to their binary approximations. DECIMAL_COMP produces the exact result because no conversion occurs when using FIXED DECIMAL variables.

These two programs illustrate a more general problem. Suppose that during a particular day, Chase Manhattan Bank processes 10,000 deposits of $3.10. Using a program with FLOAT BINARY data, $3.10 cannot be represented as a finite binary fractional expansion. Therefore it is approximated in FLOAT BINARY form as 3.099999E+00. Each addition propagates a small error into the sum, resulting in an extra $2.70 unaccounted for at the end of the day.

There are also situations where decimal arithmetic produces truncation errors that can propagate throughout computations. For example, the fraction 1/3 cannot be represented as a finite decimal fraction, and thus is approximated as

0.3333333 ...

to the maximum possible precision. Such errors only occur when explicit division operations take place.

The difficulty with FLOAT BINARY representations is that some decimal constants expressed as finite fractional expansions in FIXED DECIMAL cannot be written as finite binary fractions. PL/I necessarily truncates these during conversion to FLOAT BINARY form.

There are both advantages and disadvantages in selecting FIXED DECIMAL arithmetic instead of FLOAT BINARY. One advantage of FIXED DECIMAL arithmetic is that it guarantees there is no loss of significant digits. All digits are considered significant in a computation, so that multiplication, for example, does not truncate digits in the least significant positions. Another advantage is that FIXED DECIMAL arithmetic precludes the necessity for exponent manipulation, and the operations are relatively fast when compared to alternative decimal arithmetic formats.

The disadvantage is that you must keep track of the range of values that arithmetic operands can assume because all digits are considered significant.

## 15.2 Decimal Representation

Decimal variables and constants have both precision and scale. The precision is the number of digits in the variable or constant, while the scale is the number of digits in the fractional part. For FIXED DECIMAL variables and constants, the precision cannot exceed 15, and the scale cannot exceed the precision.

You can define the precision and scale of a variable in the variable declaration. For example,

```
declare x fixed decimal(10,3);
```

declares the variable x to have precision 10 and scale 3. The Compiler automatically derives the precision and scale of a constant by counting the number of digits in the constant, and the number of digits following the decimal point. For example, the constant

-324.76

has precision 5 and scale 2.

Internally, PL/I stores FIXED DECIMAL variables and constants as Binary Coded Decimal (BCD) pairs, where each BCD digit occupies either the high- or low-order four bits of each byte. The most significant BCD digit defines the sign of the number. A zero denotes a positive number, and a nine denotes a negative number in the 10's-complement form, as described below. Because PL/I always stores numbers into 8-bit byte locations, there can be an extra pad digit at the end of the number to align it to an even byte boundary.

For example, PL/I stores the number 83.62 as,

| 0 8 | 3 6 | 2 0 |
|---|---|---|

where each digit represents a 4-bit half-byte position in the 8-bit value. PL/I stores the leading BCD pair lowest in memory.

PL/I stores negative numbers in 10's-complement form to simplify arithmetic operations. A 10's-complement number is similar to a 2's-complement binary representation, except the complement value of each digit x is 9-x.

To derive the 10's-complement value of a number, form the complement of each digit by subtracting the digit from 9, and add 1 to the final result. Thus, the 10's complement of -2 is formed as follows:

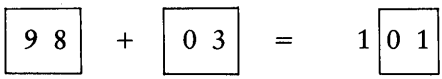
$$(9 - 2) + 1 = 7 + 1 = 8$$

PL/I adds the sign digit to the number that then appears as the single-byte value:

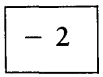
| 9 8 |
|---|

Look at an example. Suppose you want to add −2 and +3. PL/I represents these numbers as follows:

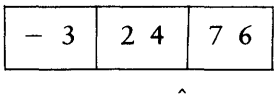



PL/I ignores the integers beyond the sign digit above, and produces the correct result 01. In the following discussion, we show negative numbers with a leading - sign, with the assumption that the internal representation is in 10's-complement form. Thus, we write the number -2 as:

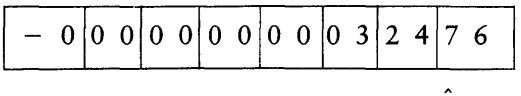



There is no need to explicitly store the decimal position in memory, because the Compiler knows the precision and scale for each variable and constant. Before each arithmetic operation, the compiled code causes the necessary alignment of the operands. In later examples, we show a decimal point position to emphasize the effect of alignment.

For example, the number −324.76 appears as:




When PL/I prepares this value for arithmetic processing, it first loads it into an 8-byte stack frame, consisting of 15 BCD digits with a high-order sign. In this case, the -324.76 is shown as:




In ordinary arithmetic, when beginning each operation you must properly align the operands for that operation and, upon completion, you must decide where the resulting decimal point appears.

In PL/I, the Compiler performs the alignment and accounts for the decimal point position, but it is useful for you to imagine what is taking place, so you can avoid overflow or underflow conditions. In some cases, you might want to force a precision

or scale change during the computation using the DECIMAL or DIVIDE built-in functions. The sample programs discussed in the following sections give examples of these functions.

## 15.3 Addition and Subtraction

In PL/I, addition and subtraction are functionally equivalent. This is because in subtraction, PL/I first forms the 10's complement of the subtrahend and then performs the addition. Given two numbers x and y, with precision and scale (p,q) and (r,s), respectively, the addition operation proceeds as follows.

First, PL/I loads the two operands onto the stack and then aligns them by shifting the operand with the smaller scale to the left until the decimal positions are the same. Given that the scale of x is greater than the scale of y, y is shifted q-s positions to the left, with zero values introduced in the least significant positions.

After alignment, y has precision r+(q−s) and scale q. PL/I signals a FIXED OVERFLOW condition if significant digits are shifted into the sign position during the alignment process.

Here is a specific example. Suppose x = 31465.2437 and y = 9343.412 so that x has precision p = 9 and scale q = 4, while y has precision r = 7 and scale s = 3. Before alignment, the numbers appear as:

```
                        |<---- p=9 ---->|
                             < q=4 >
x = + 0 0 0 0 0 0 3 1 4 6 5 2 4 3 7
                            ^

y = + 0 0 0 0 0 0 0 0 9 3 4 3 4 1 2
                              |
                           <s=3>
                    |<---- r=7 ---->|
```

PL/I aligns y with x by shifting q-s = 4−3 = 1 positions to the left, producing:

```
                    |<---- p=9 ---->|
                          < q=4 >
x = + 0 0 0 0 0 0 3 1 4 6 5 2 4 3 7
                          ^
y = + 0 0 0 0 0 0 0 9 3 4 3 4 1 2 0
                          ^
                    |         < q >|
                    |< r+(q-s) = 8 >|
```

The number of digits in the whole part of x is p-q, while the whole part of y contains r-s digits,

```
|< p-q=5 >|
 3 1 4 6 5
   9 3 4 3
  |<r-s=4>|
```

so the sum must contain p-q = 5 digits in the whole part:

```
   3 1 4 6 5
+    9 3 4 3
  -----------
   4 0 8 0 8
  |< p-q=5 >|
```

There is a possibility that some values could produce an overflow, requiring one extra digit in the whole part:

```
    9 9 9 9 9
+   9 9 9 9 9
  -----------
  1 9 9 9 9 8
 |<(p-q)+1=6>|
```

The total number of digits in the sum of x and y is the number of digits in the whole part, (p-q)+1=6, plus the number of digits in the fraction, given by q, resulting in a precision of:

$$(p-q)+1 + q = p + 1$$

Given two values x and y, of arbitrary precision and scale, you can use the specific case shown above to derive the form of the resulting precision and scale. First, the scale must be the greater of q and s, given by,

$$\max(q,s)$$

and the resulting precision must have max(q,s) fractional digits.

Second, the whole part of x contains p-q digits, while the whole part of y contains r-s digits. The result contains the larger of p-q and r-s digits, plus the fractional digits, along with one overflow digit, for a total of,

$$\max(p-q,r-s) + \max(q,s) + 1$$

digit positions.

Because the precision cannot exceed 15 digits, the resulting precision must be,

$$\min(15,\max(p-q,r-s)+\max(q,s)+1)$$

digits.

The precision and scale of the resulting addition or subtraction written as a pair (p',q') is:

$$(\ \overbrace{\min(15,\max(p-q,r-s)+\max(q,s)+1)}^{p'},\ \overbrace{\max(q,s)}^{q'}\ )$$

Using the preceding example:

```
                        <——p=9——>
                                <q=4>
        x =  + 0 0 0 0 0 0 3 1 4 6 5 2 4 3 7
                                ^
        y =  + 0 0 0 0 0 0 0 9 3 4 3 4 1 2 0
                                ^
                                <—q—>
                       |< r+(q-s) = 8 >|

    x + y =  + 0 0 0 0 0 0 4 0 8 0 8 6 5 5 7
                                ^
                                <- 4 ->
                       <——  10  ——>
```

The precision (10,4) shown in the diagram is derived using the expression,

```
           p'                           q'
  |————————————————————————————————| |————————|
( min(15,max(9-4,7-3)+max(4,3)+1), max(4,3) )
```

or:

$$( \min(15,\max(5,4)+4+1), 4 ) = (\min(15,10),4) = (10,4)$$

## 15.4 Multiplication

Evaluating the precision and scale for multiplication is simpler than addition and subtraction because PL/I does not have to align the decimal point before the multiplication. Given two operands x and y with precision and scale (p,q) and (r,s) respectively, PL/I multiplies the two operands digit by digit to produce the result.

Just as in ordinary hand calculations, the number of decimal places in the result is the sum of the scale factors q and s. The total number of digits in the result is the sum of the precisions of the two operands. To conform to the PL/I Subset G standard, PL/I includes one additional digit position in the final precision. The precision and scale of the result (p',q') is given by:

p' q'

(min(15,p+r+1),q+s)

Suppose that x = 924.5 and y = 862.33, with the precision and scale values (4,1) and (5,2):

```
x = + 0 0 0 0 0 0 0 0 0 0 0 9 2 4 5
                                  ^

y = + 0 0 0 0 0 0 0 0 0 0 0 8 6 2 3 3
                                ^
```

The product of the digits of x and y is shown with the resulting precision and scale,

```
x * y = + 0 0 0 0 0 0 7 9 7 2 2 4 0 8 5
                                  ^
                                 <3>
                  <-----10----->
```

where PL/I computes the precision and scale as:

(min(15,4+5+1),1+2) = (min(15,10),3) = (10,3)

PL/I signals the FIXEDOVERFLOW condition if the product contains more than 15 significant digits. In the previous section, where x = 31465.2437 and y = 9343.412, the product x*y has precision 17, causing FIXEDOVERFLOW.

In this particular case, you must apply the DECIMAL function to reduce the number of significant digits in either x or y. The computation is carried out as,

```
DECIMAL(x,9,3) * y
```

which loads the stack with the two following values before the multiplication takes place:

```
DECIMAL(x,9,3) = +00000031465 243
                          ^

y = +000000009343412
                    ^
```

The precision and scale of the product is:

```
+ 2 9 3 9 9 2 7 2 9 0 2 9 1 1 6
                  ^
                      <-6->
<---------15--------->
```

PL/I first computes the precision as p+r+1 = 16, and then reduces this to the maximum 15 digit precision by:

```
min(15,p+r+1) = min(15,16) = 15
```

When performing multiplication, it is your responsibility to ensure that the precisions of the operands involved do not produce overflow. You can explicitly declare the precision and scale of the variables involved in the computation, or apply the DECIMAL function to reduce the precision of a temporary result.

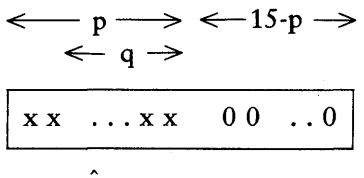
## 15.5 Division

Division is the only one of the four basic arithmetic operations that can produce truncation errors. Therefore each division operation produces a maximum precision value consisting of 15 decimal digits, and a resulting scale that depends upon the scale values of the two operands.

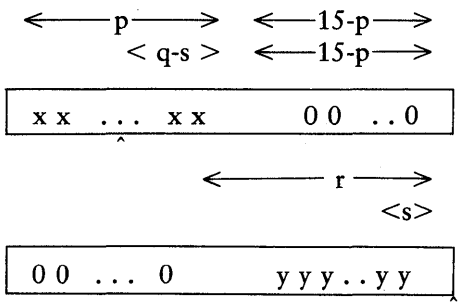
Assuming that x and y have precision and scale (p,q) and (r,s) respectively, and that x is to be divided by y, the division operation takes place as follows.

First, PL/I shifts x to the extreme left by introducing 15-p zero values on the right, leaving the dividend on the stack as:




PL/I then shifts the decimal point of x right by an amount s to properly align the decimal point in the result, producing the following operands:

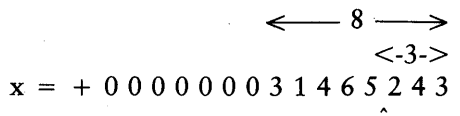



The significant digits of y then continuously divide the significant digits of x until the operation generates 15 decimal digits.

In the diagram above, the number of fractional digits produced by the division is determined by the placement of the adjusted decimal point in x. The field following the decimal point contains (q-s) plus (15-p) positions, yielding the following precision and scale for the result of the division:

(15, (q-s)+(15-p)) or (15,15-p+q-s)

Suppose x = 31465.243, and y = 9343.41, have precision and scale values of (8,3) and (6,2), respectively. The value x when loaded on the stack appears as:



PL/I then shifts the value of x to the extreme left and loads the value of y, producing the values:

```
          <------ 8 ------> <-15-8=7->
                   |< 3 >|
x = + 3 1 4 6 5 2 4 3 0 0 0 0 0 0 0
                   ^

y = + 0 0 0 0 0 0 0 0 0 9 3 4 3 4 1
                                 ^
                           |     <2>|
                           |<- 6 ->|
```

The imaginary decimal points are shifted to the right by two positions to properly align the decimal point in the result, producing:

```
          <------ 8 ------>|<-- 7 -->|
                           1
x = + 3 1 4 6 5 2 4 3 0 0 0 0 0 0 0
                   ^

y = + 0 0 0 0 0 0 0 0 0 9 3 4 3 4 1
                                 ^
                           <-- 6 -->|
```

The six significant digits of y divide the significant digits of x, and the result is:

```
          |<---------- 15 ---------->|
                          <-1+7=8->  |
x/y = + 0 0 0 0 0 0 0 3 3 6 7 6 4 0 1
                   ^
```

In this case, the precision and scale of the result is given by:

(15,(15-p+q-s) = (15,15-8+3-2) = (15,8)

The most important consideration in decimal division is generating enough digits in the fractional part for the computation being performed. This is done in two ways.

First, when aligning the dividend, PL/I pads with zeros and provides 15-p fractional digits. Thus, dividend values with small precision generate more fractional digits.

Second, if q is greater than s, then PL/I generates (q-s) additional fractional digits as shown above. If on the other hand, the dividend contains fewer fractional digits than the divisor, then q is less than s and (s-q) fractional digits are consumed.

The case of q = s occurs quite often. In this particular situation, the number of fractional digits depends entirely upon the precision of the divisor, and results in 15-p fractional digits.

You might also want to truncate or extend the result with zeros using the DIVIDE built-in function during a particular computation (see the *PL/I Language Reference Manual*, Section 4.2.5). The form of the function is,

```
DIVIDE (x,y,p,q)
```

where p and q are literal constants. They can appear as an expression or subexpression in an arithmetic computation, and have the same effect as the statement:

```
DECIMAL (x/y,p,q)
```

As before, y divides x, but the precision and scale values are forced to (p,q). PL/I carries out the computation as described above, and then shifts the resulting value by the appropriate number of digits to obtain the desired precision and scale.

*End of Section 15*

References: Sections 3.1.2, 4.2 LRM

# Section 16
# Commercial Processing

Commercial applications of PL/I use decimal calculations. The four programs in this section illustrate PL/I built-in functions, EDIT formats including Picture, and the method of breaking down a complex program into small, logically distinct procedures.

## 16.1 A Simple Loan Program

Listing 16-1 shows the LOAN1 program that computes a loan payment schedule using three input values corresponding to the loan principal (P), the yearly interest rate (i), and monthly payment (PMT). LOAN1 continuously applies the following algorithm until the remaining principal reaches zero, and the loan is paid off.

The algorithm is:

1. Each month, increase the starting principal P by an amount fixed by the interest rate.

```
P = P + (i * P)
```

2. Each month, reduce the remaining principal by the payment amount.

```
P = (P + (i * P)) - PMT
```

LOAN1 assumes that the principal does not exceed $999,999,999.99. Thus the declaration on line 12 defines P as a FIXED DECIMAL variable with precision 11 and scale 2. The payment does not exceed $9,999.99, so PMT is declared as FIXED DECIMAL with precision 6 and scale 2. Finally, LOAN1 defines the interest rate i as FIXED DECIMAL(4,2), allowing numbers as large as 99.99%. The two variables m and y correspond to the month and year, beginning at the first month of the first year.

LOAN1 reads the initial values between lines 17 and 22. In this example, LOAN1 does not perform any range checking. Thus it can accept negative values, and can process payment values that cannot pay off the loan. These checks would have to be made in a real application environment.

On each iteration, LOAN1 increases the month until it reaches the 12th month, at which point the built-in MOD function, line 26, increments the year. LOAN1 then displays the current principal P on line 32, and adds the monthly interest on the following line.

LOAN1 performs the computation on line 33. The variable i has precision and scale 4(2), while the variable P has precision and scale 11(2). Therefore, the multiplication i *P yields a temporary result with precision and scale, 15(4).

Next the division by the literal constant 1200 is required because the interest rate is expressed as a percentage (division by 100) over a one-year period (division by 12). The result of the division (i * P)/1200 has precision 15, because the constant 1200 has precision and scale, 4(0). PL/I computes precision and scale in division as (15,15-15+4-0). Finally, LOAN1 uses the built-in function ROUND to round the second decimal place, the cents position.

In the last month, if the remaining principal is less than the payment, LOAN1 performs the test on line 34. If the test is true, line 35 changes the payment to equal the principal. Line 36 prints the payment, and finally, line 37 reduces the principal by the payment using the assignment statement:

```
P = P - PMT
```

Listing 16-2 shows the output from LOAN1 using an initial loan of $500, interest rate of 14%, and payment of $22.10 per month.

```
 1 a   /*******************************************************/
 2 a   /* This program produces a schedule of loan payments   */
 3 a   /* using the following algorithm: if P = loan payment, */
 4 a   /* i = interest, and PMT is the monthly payment then   */
 5 a   /* P = (P + (i*P) - PMT).                              */
 6 a   /*******************************************************/
 7 a   loan1:
 8 b       procedure options(main);
 9 b       declare
10 b       m    fixed binary,
11 b       y    fixed binary,
12 b       P    fixed decimal(11,2),
13 b       PMT fixed decimal(6,2),
14 b       i    fixed decimal(4,2);
15 b
16 c       do while('1'b);
17 c          put skip list('Principal ');
18 c          get list(P);
19 c          put list('Interest  ');
20 c          get list(i);
21 c          put list('Payment   ');
22 c          get list(PMT);
23 c          m = 0;
24 c          y = 0;
25 d          do while (P > 0);
26 d             if mod(m,12) = 0 then
27 e             do;
28 e                y = y + 1;
29 e                put skip list('Year',y);
30 e             end;
31 d             m = m + 1;
32 d             put skip list(m,P);
33 d             P = P + round( i * P / 1200, 2);
34 d             if P < PMT
35 d                then PMT = P;
36 d             put list(PMT);
37 d             P = P - PMT;
38 d          end;
39 c       end;
40 b
41 b   end loan1;
```

**Listing 16-1. The LOAN1 Program**

```
A>loan1

Principal 500
Interest  14
Payment   22.10

Year          1
        1             500.00     22.10
        2             483.73     22.10
        3             467.27     22.10
        4             450.62     22.10
        5             433.78     22.10
        6             416.74     22.10
        7             399.50     22.10
        8             382.06     22.10
        9             364.42     22.10
       10             346.57     22.10
       11             328.51     22.10
       12             310.24     22.10
Year          2
       13             291.76     22.10
       14             273.06     22.10
       15             254.15     22.10
       16             235.02     22.10
       17             215.66     22.10
       18             196.08     22.10
       19             176.27     22.10
       20             156.23     22.10
       21             135.95     22.10
       22             115.44     22.10
       23              94.69     22.10
       24              73.69     22.10
Year          3
       25              52.45     22.10
       26              30.96     22.10
       27               9.22     22.10
Principal ^C
A>
```

**Listing 16-2. Output from the LOAN1 Program**

## 16.2 Ordinary Annuity

Listing 16-3 shows the ANNUITY program. Given the interest rate (i) and two of three values, ANNUITY computes either the present value (PV), the payment (PMT), or the number of pay periods (n) for an ordinary annuity.

ANNUITY contains one main loop between lines 35 and 80 which reads the present value, payment, and yearly interest from the console. On each iteration, you enter two nonzero values and one zero value, then ANNUITY computes the value of the variable that you enter as zero. ANNUITY retains the values on each loop so that you can enter a comma if you do not want to change the value. In this example, ANNUITY does not check that the input values are in the proper range.

```
 1 a   /*******************************************************/
 2 a   /* This program computes either the present value(PV), */
 3 a   /* the payment(PMT), or the number of periods in an    */
 4 a   /* ordinary annuity.                                   */
 5 a   /*******************************************************/
 6 a   annuity:
 7 b       procedure options(main);
 8 b       %replace
 9 b          clear by '^z',
10 b          true  by '1'b;
11 b       declare
12 b          PMT fixed decimal(7,2),
13 b          PV  fixed decimal(9,2),
14 b          IP  fixed decimal(6,6),
15 b          x   float binary,
16 b          yi  float binary,
17 b          i   float binary,
18 b          n   fixed;
19 b
20 b       declare
21 b          ftc entry(float binary(24))
22 b              returns(character(17) varying);
23 b
24 b       put list (clear,'^i^iO R D I N A R Y   A N N U I T Y');
25 b       put skip(2) list
26 b       ('^iEnter Known Values, or 0, on Each Iteration');
27 b
28 b       on error
29 c          begin;
30 c            put skip list('^iInvalid Data, Re-enter');
31 c            goto retry;
32 c          end;
33 b
```

**Listing 16-3. The ANNUITY Program**

```
34 b        retry:
35 c      do while (true);
36 c         put skip(3) list('^iPresent Value ');
37 c         get list(PV);
38 c         put list('^iPayment       ');
39 c         get list(PMT);
40 c         put list('^iInterest Rate ');
41 c         get list(yi);
42 c         i = yi / 1200;
43 c         put list('^iPay Periods   ');
44 c         get list(n);
45 c
46 c         if PV = 0 | PMT = 0 then
47 c            x = 1 - 1/(1+i)*/n;
48 c
49 c         /*****************************/
50 c         /* compute the present value */
51 c         /*****************************/
52 c         if PV = 0 then
53 c           do;
54 d            PV = PMT * dec(ftc(x/i),15,6);
55 d            put edit('^iPresent Value is ',PV)
56 d                    (a,p'$$$,$$$,$$$V.99');
57 d           end;
58 c
59 c         /************************/
60 c         /* compute the payment */
61 c         /************************/
62 c          if PMT = 0 then
63 d           do;
64 d             PMT = PV * dec(ftc(i/x),15,8);
65 d             put edit('^iPayment is ',PMT)
66 d                     (a,p'$$,$$$,$$$V.99');
67 d           end;
68 c
69 c         /*****************************/
70 c         /* compute number of periods */
71 c         /*****************************/
72 c         if n = 0 then
73 d           do;
74 d             IP = ftc(i);
75 d             x = char(PV * IP / PMT);
76 d             n = ceil ( - log(1-x)/log(1+i) );
77 d             put edit('^i',n,' Pay Periods')
78 d                     (a,p'ZZZ9',a);
79 d           end;
80 c       end;
81 b
82 b    end annuity;
```

**Listing 16-3. (continued)**

Listing 16-4 shows an interaction with the ANNUITY program in which several different values are used as input.

```
A>annuity
              O R D I N A R Y   A N N U I T Y

      Enter Known Values, or 0, on Each Iteration


Present Value 32000
Payment       0
Interest Rate 8.75
Pay Periods   360
Payment is       $251.74

Present Value ,
Payment       0
Interest Rate ,
Pay Periods   240
Payment is       $282.78

Present Value 0
Payment       ,
Interest Rate ,
Pay Periods   ,
Present Value is     $31,998.87

Present Value 32000
Payment       ,
Interest Rate ,
Pay Periods   0
 240 Pay Periods

Present Value ^C
A>
```

**Listing 16-4. Interaction with the ANNUITY Program**

### 16.2.1 Mixed Data Types

ANNUITY uses both FLOAT BINARY and FIXED DECIMAL data because it must perform a mixture of decimal arithmetic calculations and analytic function evaluations. The variables used throughout the program are defined between lines 12 and 18 as follows:

- PMT holds the payment value, is declared as FIXED DECIMAL(7,2), and can be as large as $99,999.99.

- PV holds the present value, is declared as FIXED DECIMAL(9,2), and can be as large as $99,999,999.99.
- The variable IP holds the interest rate for a one month period, and is declared as FIXED DECIMAL with six decimal places.
- The variable n holds the number of payment periods, is declared as FIXED BINARY, and can range from 1 to 32767.
- The variables x, yi, and i are FLOAT BINARY numbers used during the computations to approximate decimal numbers with 7 decimal places.

ANNUITY computes the unknown value using the equations shown below, rather than the iteration. ANNUITY assumes the interest rate is greater than zero.

First, the present value is given by:

$$PV = PMT \frac{1 - \frac{1}{(1 + i)^n}}{i} \qquad (1)$$

Transposing equation (1) gives:

$$PMT = PV \frac{i}{1 - \frac{1}{(1 + i)^n}} \qquad (2)$$

Finally, solving for n gives:

$$n = -\frac{\text{Log}\left(1 - PV\left(\frac{i}{PMT}\right)\right)}{\text{Log}(1 + i)} \qquad (3)$$

The following expression appears in both equations (1) and (2):

```
1 - 1/(1 + i) ** n
```

Therefore, ANNUITY stores this value in the variable x, line 47, and uses it when evaluating PV and PMT. x is only an approximation of the decimal value given by this expression.

### 16.2.2 Evaluating the Present Value PV

If you enter a zero value for PV, then ANNUITY executes the DO-group between lines 53 and 57, and computes PV as:

```
PV = PMT * dec(ftc(x/i),15,6);
```

Line 20 declares ftc as an external subroutine. It is part of the PL/I Run-time Subroutine Library (RSL), so ANNUITY only needs to declare it as an entry constant to use it.

The division x/i produces a FLOAT BINARY temporary result that ftc then converts from FLOAT to CHARACTER form. For example, suppose that x/i produces the value 3.042455E+01. Then ftc(x/i) returns 30.42455 which is acceptable for conversion to decimal. If PL/I cannot convert the floating-point argument to a 15-digit decimal number, ftc signals the ERROR(1) condition, indicating a conversion error.

Finally, the built-in DECIMAL function is applied to the character string to convert it to a specific precision and scale, 15(6). When this is done, the multiplication and subsequent assignment to PMT takes place.

How is this particular value for precision and scale decided? To answer the question, first consider a restricted form of the same program,

```
declare
  PMT  fixed decimal(7,2),
  PV   fixed decimal(9,2),
  Q    fixed decimal(u,v);
PV = PMT * Q;
```

where you must decide on the appropriate constant values for u and v.

PV has precision and scale 9(2), and thus there must be seven digits in the whole part and two digits in the fraction. PL/I generates the full seven digits in the whole part if the product PMT * Q results in any of the precision and scale values:

(9,2) (10,3) (11,4) (12,5) (13,6) (14,7) (15,8)

The assignment to PV truncates any fractional digits beyond the second decimal place. Because PMT has precision and scale (7,2), you can choose Q with a precision and scale of (15,6). Then the multiplication produces a result with precision and scale,

$$(\min(15,7+15+1),2+6) = (15,8)$$

according to the rules stated previously.

Given an expression with precision and scale values as shown below,

```
  a     =    b     *     c
(p,q)        (r,s)       (u,v)
```

where p, q, r, and s are constants, you can set the precision and scale of c to:

$$u = 15 \qquad v = 15 - (p + q - s)$$

Thus, using the values shown in the original program, the precision and scale of Q becomes:

$v = 15 - (9 + 2 - 2) = 8$, or $(u,v) = (15,6)$

### 16.2.3 Evaluating the Payment PMT

If you enter a nonzero present value for PV and a zero value for the payment PMT, then ANNUITY enters the DO-group beginning at line 63 and computes the value of PMT as:

```
PMT = PV * dec (ftc(i/x),15,8);
```

The computation uses essentially the same technique as shown in the previous example. You must decide the precision and scale of the second operand in the multiplication. You are really concerned only with the value of the scale because the precision can be taken as 15.

Using the analysis shown above, evaluate the form,

```
  a     =    b     *     c
(7,2)        (9,2)       (15,v)
```

and determine the value for v:

$$v = 15 - (p + q - s) = 15 - (7 + 2 - 2) = 8$$

### 16.2.4 Evaluating the Number of Periods n

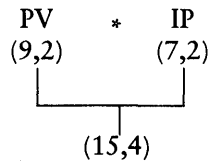
When you enter nonzero values for PV and PMT, but set the number of periods to zero, ANNUITY executes the DO-group beginning on line 73 to compute n. The assignment on line 74 first changes the interest for a monthly period from FLOAT BINARY to FIXED DECIMAL.

Next, the assignment on line 75,

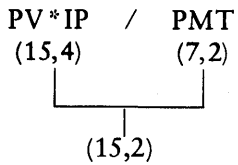
```
x = char(PV * IP / PMT);
```

first computes the partial decimal result PV * IP / PMT, then converts the result to CHARACTER, and then to FLOAT BINARY through the assignment to x.

The multiplication PV * IP produces a temporary result with the precision and scale:




The temporary result is now divided by PMT and results in another temporary result with the following precision and scale,




because, according to the rules for division,

$$(15,15-p+q-s) = (15,15-15+4-2) = (15,2)$$

thus providing two decimal places in the computation.

The intermediate conversion to CHARACTER form is necessary because otherwise PL/I would first convert the intermediate result to FIXED BINARY, and then to FLOAT BINARY, resulting in truncation of the fraction. This sequence of conversions is necessary to maintain compatibility with the full language.

If required, you could generate additional fractional digits by applying the DECIMAL built-in function following the multiplication,

```
x = char( dec( PV*P, 11,4 ) / PMT);
```

and produce a quotient with precision and scale:

```
(15,15-11+4-2) = (15,6)
```

ANNUITY uses the value x in the expression on line 76 to compute the number of payment periods, and applies the CEIL function to the result so that any partial month is treated as a full month in the payment period analysis.

Finally, ANNUITY uses the Picture edit format to write out the values of PV, PMT and n.

## 16.3 Loan Payment Schedule Format

The LOAN2 program shown in Listing 16-5 is essentially the same as that presented in Section 16.1, but it has a more elaborate analysis and display format. LOAN2 uses an algorithm similar to that described in Section 16.1. The main processing occurs between lines 101 and 136, where the program increases the initial principal by the monthly interest, and reduces it by the monthly payment until the principal becomes zero.

The four listings that follow the discussion of the program show several examples of interaction with LOAN2.

Listing 16-6 shows a minimal display corresponding to a loan of $3000 at a 14% interest rate with a payment of $144.03. Assume an inflation rate of 0% with a starting payment on 11/80, and end-of-year taxes due in December.

The display shows the principal, interest in December, monthly payment, amount paid toward principal in December, and amount of interest paid in the last month of the fiscal year.

Listing 16-7 shows another execution using the same values as the first time, but using a display level of 1. The output also contains the yearly interest paid on the loan for each fiscal year that would be deducted from the taxable income.

Listing 16-8 uses the same initial values of the previous examples, but provides a full display of the monthly principal, interest, monthly payment, payment applied to the principal, and interest payment.

Listing 16-9 also shows the same loan and interest rate with an adjustment in dollar value due to inflation. This example assumes the inflation rate of 10%, so that all amounts are scaled to the value of the dollar at the time the loan is issued.

For tax reporting, the display showing the total interest paid at the end of each year is not scaled, and thus does not match the sum of the interest paid during the year. If we assume a 0% inflation rate, the total loan payment is 3,456.97, taken from the previous output.

But if we assume an inflation rate of 10%, the total cost of the loan in dollars today is,

```
    2,457.00
+     374.25
-------------
    2,831.25
```

resulting in a net gain of 68.75 over a two year period!

```
 1 a   /*****************************************************/
 2 a   /* This program computes a schedule of loan payments */
 3 a   /* using an elaborate analysis and display format.   */
 4 a   /* It contains five internal procedures: DISPLAY,    */
 5 a   /* SUMMARY, CURRENT_YEAR, HEADER, and LINE.          */
 6 a   /*****************************************************/
 7 a   loan2:
 8 b       procedure options(main);
 9 b       %replace
10 b          true   by '1'b,
11 b          false  by '0'b,
12 b          clear  by '^z';
13 b
```

**Listing 16-5. The LOAN2 Program**

```
14 b        declare
15 b           end bit(1),
16 b           m   fixed binary,
17 b           sm  fixed binary,
18 b           y   fixed binary,
19 b           sy  fixed binary,
20 b           fm  fixed binary,
21 b           dl  fixed binary,
22 b           P   fixed decimal(10,2),
23 b           PV  fixed decimal(10,2),
24 b           PP  fixed decimal(10,2),
25 b           PL  fixed decimal(10,2),
26 b           PMT fixed decimal(10,2),
27 b           PMV fixed decimal(10,2),
28 b           INT fixed decimal(10,2),
29 b           YIN fixed decimal(10,2),
30 b           IP  fixed decimal(10,2),
31 b           yi  fixed decimal(4,2),
32 b           i   fixed decimal(4,2),
33 b           INF fixed decimal(4,3),
34 b           ci  fixed decimal(15,14),
35 b           fi  fixed decimal(7,5),
36 b           ir  fixed decimal(4,2);
37 b
38 b        declare
39 b           name character(14) varying static initial('$con'),
40 b           output file;
41 b
42 b        put list(clear,'^i^iS U M M A R Y   O F   P A Y M E N T S');
43 b
44 b        on undefinedfile(output)
45 c          begin;
46 c             put skip list('^i^icannot write to',name);
47 c             goto open_output;
48 c          end;
49 b
50 b        open_output:
51 b        put skip(2) list('^i^iOutput File Name ');
52 b        get list(name);
53 b        if name = '$con' then
54 b           open file(output) title('$con') print pagesize(0);
55 b        else
56 b           open file(output) title(name) print;
57 b
```

**Listing 16-5. (continued)**

```
 58 b       on error
 59 c        begin;
 60 c          put skip list('^i^iBad Input Data, Retry');
 61 c          goto retry;
 62 c        end;
 63 b
 64 b     retry:
 65 c     do while(true);
 66 c        put skip(2) list('^i^iPrincipal      ');
 67 c        get list(PV);
 68 c        P = PV;
 69 c        put list('^i^iInterest       ');
 70 c        get list(yi);
 71 c        i = yi;
 72 c        put list('^i^iPayment        ');
 73 c        get list(PMV);
 74 c        PMT = PMV;
 75 c        put list('^i^i%Inflation     ');
 76 c        get list(ir);
 77 c        fi = 1 + ir/1200;
 78 c        ci = 1.00;
 79 c        put list('^i^iStarting Month ');
 80 c        get list(sm);
 81 c        put list('^i^iStarting Year  ');
 82 c        get list(sy);
 83 c        put list('^i^iFiscal Month   ');
 84 c        get list(fm);
 85 c        put edit('^i^iDisplay Level  ',
 86 c                 '^i^iYr Results : 0 ',
 87 c                 '^i^iYr Interest: 1 ',
 88 c                 '^i^iAll Values : 2 ')
 89 c             (skip,a);
 90 c        get list(dl);
 91 c        if dl < 0| dl > 2 then
 92 c           signal error;
 93 c        m = sm;
 94 c        y = sy;
 95 c        IP = 0;
 96 c        PP = 0;
 97 c        YIN = 0;
 98 c        if name ^= '' then
 99 c           put file(output) page;
100 c        call header();
```

**Listing 16-5. (continued)**

```
101 d     do while (P > 0);
102 d        end = false;
103 d        INT = round ( i * P / 1200, 2 );
104 d        IP  = IP + INT;
105 d        PL  = P;
106 d        P   = P  + INT;
107 d        if P < PMT then
108 d           PMT = P;
109 d        P   = P - PMT;
110 d        PP  = PP + (PL - P);
111 d        INF = ci;
112 d        ci = ci / fi;
113 d        if P = 0 | dl > 1 | m = fm then
114 e           do;
115 e              put file(output) skip
116 e                  edit('|',100*m+y) (a,p'99/99');
117 e              call display(PL * INF, INT * INF,
118 e              PMT * INF, PP * INF, IP * INF);
119 e           end;
120 d        if m = fm & dl > 0 then
121 d           call summary();
122 d        m = m + 1;
123 d        if m > 12 then
124 e           do;
125 e              m = 1;
126 e              y = y + 1;
127 e              if y > 99 then
128 e                 y = 0;
129 e           end;
130 d     end;
131 c     if dl = 0 then
132 c        call line();
133 c     else
134 c       if ^end then
135 c          call summary();
136 c  end retry;
```

**Listing 16-5. (continued)**

```
137 b   /*****************************************************/
138 b   /* This procedure performs the output of the actual */
139 b   /* parameters passed to it by the main part of the  */
140 b   /* program.                                          */
141 b   /*****************************************************/
142 b   display:
143 c      procedure(a,b,c,d,e);
144 c      declare
145 c         (a,b,c,d,e) fixed decimal(10,2);
146 c
147 c      put file (output) edit
148 c          ('|',a,'|',b,'|',c,'|',d,'|',e,'|')
149 c           (a,2(2(p'$zz,zzz,zz9v.99',a),
150 c          p'$zzz,zz9.v99',a));
151 c   end display;
152 b
153 b   /*************************************************/
154 b   /* This procedure computes the summary of yearly */
155 b   /* interest.                                     */
156 b   /*************************************************/
157 b   summary:
158 c      procedure;
159 c      end = true;
160 c      call current_year(IP-YIN);
161 c      YIN = IP;
162 c   end summary;
163 b
164 b   /*****************************************************/
165 b   /* This procedure computes the interest paid during */
166 b   /* current year.                                     */
167 b   /*****************************************************/
168 b   current_year:
169 c      procedure(I);
170 c      declare
171 c         yp fixed binary,
172 c         I fixed decimal(10,2);
173 c      yp = y;
174 c      if fm < 12 then
175 c         yp = yp - 1;
176 c      call line();
177 c      put skip file(output) edit
178 c          ('|','Interest Paid During ''',yp,'-''',y,' is ',I,'|')
179 c          (a,x(15),2(a,p'99'),a,p'$$$,$$$,$$9v.99',x(16),a);
180 c      call line();
181 c   end current_year;
182 b
```

**Listing 16-5. (continued)**

```
183 b   /*****************************************************/
184 b   /* This procedure defines and prints out an elaborate */
185 b   /* header format.                                     */
186 b   /*****************************************************/
187 b ┌ header:
188 c │    procedure;
189 c │    put file(output) list(clear);
190 c │    call line();
191 c │    put file(output) skip edit
192 c │        ('¦','L O A N   P A Y M E N T   S U M M A R Y','¦')
193 c │        (a,x(19));
194 c │    call line();
195 c │    put file(output) skip edit
196 c │        ('¦','Interest Rate',yi,'%','Inflation Rate',ir,'%','¦')
197 c │        (a,x(15),2(a,p'b99v.99',a,x(6)),x(9),a);
198 c │    call line();
199 c │    put file(output) skip edit
200 c │    ('¦Date ¦',' Principal   ¦','Plus  Interest¦',' Payment  ¦',
201 c │         'Principal Paid¦','Interest Paid ¦') (a);
202 c │    call line();
203 c └ end header;
204 b
205 b   /*****************************************************/
206 b   /* This procedure prints out a series of dashed lines. */
207 b   /*****************************************************/
208 b ┌ line:
209 c │    procedure;
210 c │    declare
211 c │       i fixed bin;
212 c │    put file(output) skip edit
213 c │        ('-------','------------',
214 c │        ('---------------' do i = 1 to 4)) (a);
215 c └ end line;
216 b
217 b
218 b  end loan2;
```

**Listing 16-5. (continued)**

### 16.3.1 Variable Declarations

Starting on line 14, LOAN2 declares several data items:

- PV present value, initial principal
- yi yearly interest rate
- PMV monthly payment
- ir yearly inflation rate
- sm starting month of payment (1-12)
- sy starting year of payment (0-99)
- fm fiscal month, end of fiscal year (1-12)
- dl display level (0-2)

LOAN2 declares the initial principal and payment variables as FIXED DECIMAL (10,2), allowing values as large as $99,999,999.99.

It also allows the yearly interest rate and yearly inflation rate to be as large as 99.99.

The month and year variables, sm, sy, and fm are FIXED BINARY and LOAN2 assumes that these variables properly represent month and year values.

The variable dl is the display level and defines the amount of information LOAN2 displays during a particular iteration of the program. That is, 0 produces an abbreviated display, 1 produces additional information, and 2 gives the full trace.

LOAN2 also declares several other variables used throughout the program:

- P initially set to PV, but changes during execution
- PP total principal paid
- PL principal for current line, holds P for display purposes
- PMT payment initially set to PMV; changes during execution
- INT computed interest during current month
- YIN interest at beginning of current year
- IP total interest paid
- i interest rate, initialized to yi
- INF percent devaluation of original dollar due to inflation
- ci current devaluation due to inflation
- fi factor for computing current inflation

P and PMT are working variables for the principal and payment, so that the program does not destroy the original variables PV and PMV during the computations. If you enter a comma for subsequent input requests, LOAN2 retains the previous value.

### 16.3.2 Program Execution

The program execution begins on line 42 with a clear screen character for the Lear-Siegler ADM-3A CRT. This control character is defined in the %REPLACE statement on line 12. If you are not using an ADM-3A, you can substitute the proper character and recompile the program.

LOAN2 sets an ON-condition to trap possible errors in the OPEN statement, lines 54 to 56, and then prompts for the report output filename. LOAN2 initializes the variable name to the value $con, and if you enter a comma rather than a file or device name, LOAN2 assumes the console as the output device.

If you enter either a comma or the name $con as the output filename, then the OPEN statement, line 54, opens the console with a zero page size. This means that the run-time system does not issue any form-feeds at the end of each logical page. Otherwise, LOAN2 opens the output file or device as a normal PRINT file so that the run-time system places form-feeds into the output file or sends them to the physical output device, usually the printer, denoted by $lst.

The ON condition set at line 58 traps any occurrence of the ERROR condition, including ERROR(1), which indicates a data conversion error. LOAN2 also programmatically signals invalid data on line 92 if the value of dl is out of range.

LOAN2 does not contain a complete set of routines for error checking. To make the program commercially functional, it should signal errors for all other invalid input data items, such as a negative interest rate. Furthermore, out-of-bounds computations should signal a FIXED OVERFLOW condition.

Beginning on line 67, LOAN2 reads a set of input values, and then initializes the variables for each set of input values beginning on line 93. The PUT LIST statement on line 99 executes a page eject if the output file is not the console. Line 100 then prints a page header by calling the HEADER subroutine. You should compare the formatting statements in the header subroutine with the output values shown in the output listings.

The main processing takes place in the DO-group beginning at line 101, that executes repeatedly until the principal is reduced to zero. The variable end indicates whether an end-of-year summary has been printed, line 159, and thus avoids the possibility of printing a duplicate summary, line 134.

On lines 103 and 104, LOAN2 computes the monthly interest INT for the current principal P and sums it in IP. LOAN2 saves the current principal for later display in PL, and then adds the monthly interest to the principal. If the payment exceeds the remaining principal on line 107, LOAN2 reduces the payment to cover this remainder. It then reduces the principal by the payment amount, eventually producing a zero value if the original payment is sufficient to pay off the loan. Then on lines 111 and 112 LOAN2 sums the total principal paid and computes the inflation rate.

### 16.3.3 Display Formats

The decision logic for displaying the current computation is somewhat complicated because LOAN2 has three display formats. If it is the last iteration, the principal P is zero, you select the full display format, dl > 1, or the current month is the end of the fiscal year, m = fm, then LOAN2 writes the current computation between lines 114 and 118.

The Picture format p'99/99' displays the month and year, where 100*m+y produces a four-digit number to match this format. For example, if m = 11 and y = 64, then,

```
100 * m + y = 100 * 11 + 64 = 1164
```

1164 appears as 11/64, when printed using the given Picture format.

The DISPLAY subroutine actually performs the output function, based upon the six actual parameters listed in the CALL statement on line 117. The main program first adjusts each argument, by the current inflation rate INF, and then passes it to DISPLAY. If the inflation rate is set to 0%, the value of INF is 1.00 at this point in the computation.

The body of the display subroutine, listed between lines 142 and 151 could be included in the line subroutine because there is only one call to display. However, display illustrates FIXED DECIMAL parameter passing mechanisms and serves to break the program into smaller, more readable, segments. Again, you should compare the format specifications in the display subroutine with the actual program output.

The statement on line 120 then checks for the end of fiscal year, m = fm, and, if the display mode is either 1 or 2, LOAN2 prints a yearly interest summary using the summary subroutine. Summary in turn, calls the current_year subroutine to write the yearly interest paid, IP-YIN. The assignment on line 161 retains the base value for the next year's display in YIN.

If the fiscal year does not end in December, fm< 12, current_year splits the interest rate payment between two calendar years, yp = y - 1. Again, you could combine current_year with the summary subroutine without changing the overall program logic.

The end of the main loop, between lines 131 and 136, contains statements that finalize the report. If you select the abbreviated display format, dl = 0, the CALL statement on line 132 invokes LINE and prints a line of dashes to complete the display. Otherwise, LOAN2 checks to ensure there have been intervening output lines ( end). If there have been, it prints an interest summary on line 130. Finally, control returns to the top of the DO-group, and LOAN2 reads additional input parameters.

```
      A>loan2
403
404   S U M M A R Y   O F   P A Y M E N T S
405
406   Output File Name ,
401   Principal        3000
402   Interest         14
403   Payment          144.03
404   %Inflation       0
405   Starting Month   11
406   Starting Year    80
407   Fiscal Month     12
408
409   Display Level
410   Yr Results : 0
411   Yr Interest: 1
412   All Values : 2 0
413
414
415   ------------------------------------------------------------------------
416   |              L O A N   P A Y M E N T   S U M M A R Y                  |
417   ------------------------------------------------------------------------
418   |           Interest Rate 14.00%       Inflation Rate 00.00%            |
419   ------------------------------------------------------------------------
420   |Date |Principal  |Plus Interest|Payment  |Principal Paid|Interest Paid|
421   ------------------------------------------------------------------------
422   |12/80|$  2,890.97|$       33.73|$ 144.03|$       219.33|$       68.73|
423   |12/81|$  1,479.02|$       17.26|$ 144.03|$     1,647.75|$      388.67|
424   |11/82|$      0.25|$        0.00|$   0.25|$     3,000.00|$      456.97|
425   ------------------------------------------------------------------------
```

**Listing 16-6. First Interaction with LOAN2**

```
Output File Name ,
Principal        ,
Interest         ,
Payment          ,
%Inflation       ,
Starting Month   ,
Starting Year    ,
Fiscal Month     ,

Display Level
Yr Results : 0
Yr Interest: 1
All Values : 2 1
```

```
-------------------------------------------------------------------------
|              L O A N   P A Y M E N T   S U M M A R Y                  |
-------------------------------------------------------------------------
|           Interest Rate 14.00%      Inflation Rate 00.00%             |
-------------------------------------------------------------------------
|Date |Principal  |Plus Interest |Payment |Principal Paid|Interest Paid|
-------------------------------------------------------------------------
|12/80|$  2,890.97|$        33.73|$ 144.03|$       219.33|$       68.73|
-------------------------------------------------------------------------
|           Interest Paid During `80-`80 is          $68.73              |
-------------------------------------------------------------------------
|12/81|   1,479.02|$        17.26|$ 144.03|$    1,647.75|$      368.67 |
-------------------------------------------------------------------------
|           Interest Paid During `81-`81 is         $299.94              |
-------------------------------------------------------------------------
|11/82|$      0.25|$         0.00|$   0.25|$    3,000.00|$      456.97 |
-------------------------------------------------------------------------
|           Interest Paid During `82-`82 is          $88.30              |
-------------------------------------------------------------------------
```

**Listing 16-7. Second Interaction with LOAN2**

```
Output File Name ,
Principal        ,
Interest         ,
Payment          ,
%Inflation       ,
Starting Month   ,
Starting Year    ,
Fiscal Month     ,

Display Level
Yr Results : 0
Yr Interest: 1
All Values : 2 2
```

| LOAN PAYMENT SUMMARY | | | | | |
|---|---|---|---|---|---|
| Interest Rate 14.00% | | | Inflation Rate 00.00% | | |
| Date | Principal | Plus Interest | Payment | Principal Paid | Interest Paid |
| 11/80 | $ 3,000.00 | $ 35.00 | $ 144.03 | $ 109.03 | $ 35.00 |
| 12/80 | $ 2,890.97 | $ 33.73 | $ 144.03 | $ 219.33 | $ 68.73 |
| Interest Paid During '80-'80 is | | | | $68.73 | |
| 01/81 | $ 2,780.67 | $ 32.44 | $ 144.03 | $ 330.92 | $ 101.17 |
| 02/81 | $ 2,669.08 | $ 31.14 | $ 144.03 | $ 443.81 | $ 132.31 |
| 03/81 | $ 2,556.19 | $ 29.82 | $ 144.03 | $ 558.02 | $ 162.13 |
| 04/81 | $ 2,441.98 | $ 28.49 | $ 144.03 | $ 673.56 | $ 190.62 |
| 05/81 | $ 2,326.44 | $ 27.14 | $ 144.03 | $ 790.45 | $ 217.76 |
| 06/81 | $ 2,209.55 | $ 25.78 | $ 144.03 | $ 908.70 | $ 243.54 |
| 07/81 | $ 2,091.30 | $ 24.40 | $ 144.03 | $ 1,028.33 | $ 267.94 |
| 08/81 | $ 1,971.67 | $ 23.00 | $ 144.03 | $ 1,149.36 | $ 290.94 |
| 09/81 | $ 1,850.64 | $ 21.59 | $ 144.03 | $ 1,271.80 | $ 312.53 |
| 10/81 | $ 1,728.20 | $ 20.16 | $ 144.03 | $ 1,395.67 | $ 332.69 |
| 11/81 | $ 1,604.33 | $ 18.72 | $ 144.03 | $ 1,520.98 | $ 351.41 |
| 12/81 | $ 1,479.02 | $ 17.26 | $ 144.03 | $ 1,647.75 | $ 368.67 |
| Interest Paid During '81-'81 is | | | | $299.94 | |
| 01/82 | $ 1,352.25 | $ 15.78 | $ 144.03 | $ 1,776.00 | $ 384.45 |
| 02/82 | $ 1,224.00 | $ 14.28 | $ 144.03 | $ 1,905.75 | $ 398.73 |
| 03/82 | $ 1,094.25 | $ 12.77 | $ 144.03 | $ 2,037.01 | $ 411.50 |
| 04/82 | $ 962.99 | $ 11.23 | $ 144.03 | $ 2,169.81 | $ 422.73 |
| 05/82 | $ 830.19 | $ 9.69 | $ 144.03 | $ 2,304.15 | $ 432.42 |
| 06/82 | $ 695.85 | $ 8.12 | $ 144.03 | $ 2,440.06 | $ 440.54 |
| 07/82 | $ 559.94 | $ 6.53 | $ 144.03 | $ 2,577.56 | $ 447.07 |
| 08/82 | $ 422.44 | $ 4.93 | $ 144.03 | $ 2,716.66 | $ 452.00 |
| 09/82 | $ 283.34 | $ 3.31 | $ 144.03 | $ 2,857.38 | $ 455.31 |
| 10/82 | $ 142.62 | $ 1.66 | $ 144.03 | $ 2,999.75 | $ 456.97 |
| 11/82 | $ 0.25 | $ 0.00 | $ 0.25 | $ 3,000.00 | $ 456.97 |
| Interest Paid During '82-'82 is | | | | $88.30 | |

**Listing 16-8. Third Interaction with LOAN2**

```
Output File Name ,
Principal        ,
Interest         ,
Payment          ,
%Inflation       10
Starting Month   ,
Starting Year    ,
Fiscal Month     10

Display Level
Yr Results : 0
Yr Interest: 1
All Values : 2 2
```

L O A N  P A Y M E N T  S U M M A R Y

Interest Rate 14.00%  Inflation Rate 10.00%

| Date | Principal | Plus Interest | Payment | Principal Paid | Interest Paid |
|---|---|---|---|---|---|
| 11/80 | $ 3,000.00 | $ 35.00 | $ 144.03 | $ 109.03 | $ 35.00 |
| 12/80 | $ 2,864.95 | $ 33.42 | $ 142.73 | $ 217.35 | $ 68.11 |
| 01/81 | $ 2,733.39 | $ 31.88 | $ 141.58 | $ 325.29 | $ 99.45 |
| 02/81 | $ 2,602.35 | $ 30.36 | $ 140.42 | $ 432.71 | $ 129.00 |
| 03/81 | $ 2,471.83 | $ 28.83 | $ 139.27 | $ 539.60 | $ 156.77 |
| 04/81 | $ 2,341.85 | $ 27.32 | $ 138.12 | $ 645.94 | $ 182.80 |
| 05/81 | $ 2,212.44 | $ 25.81 | $ 136.97 | $ 751.71 | $ 207.08 |
| 06/81 | $ 2,083.60 | $ 24.31 | $ 135.82 | $ 856.90 | $ 229.65 |
| 07/81 | $ 1,955.36 | $ 22.81 | $ 134.66 | $ 961.48 | $ 250.52 |
| 08/81 | $ 1,829.70 | $ 21.34 | $ 133.65 | $ 1,066.60 | $ 269.99 |
| 09/81 | $ 1,702.58 | $ 19.86 | $ 132.50 | $ 1,170.05 | $ 287.52 |
| 10/81 | $ 1,576.11 | $ 18.38 | $ 131.35 | $ 1,272.85 | $ 303.41 |
| Interest Paid During '80-'81 is | | | | $332.69 | |
| 11/81 | $ 1,451.91 | $ 16.94 | $ 130.34 | $ 1,376.48 | $ 318.02 |
| 12/81 | $ 1,326.68 | $ 15.48 | $ 129.19 | $ 1,478.03 | $ 330.69 |
| 01/82 | $ 1,203.50 | $ 14.04 | $ 128.18 | $ 1,580.64 | $ 342.16 |
| 02/82 | $ 1,079.56 | $ 12.59 | $ 127.03 | $ 1,680.87 | $ 351.67 |
| 03/82 | $ 957.46 | $ 11.17 | $ 126.02 | $ 1,782.38 | $ 360.06 |
| 04/82 | $ 835.87 | $ 9.74 | $ 125.01 | $ 1,883.39 | $ 366.92 |
| 05/82 | $ 714.79 | $ 8.34 | $ 124.00 | $ 1,983.87 | $ 372.31 |
| 06/82 | $ 594.25 | $ 6.93 | $ 123.00 | $ 2,083.81 | $ 376.22 |
| 07/82 | $ 474.26 | $ 5.53 | $ 121.99 | $ 2,183.19 | $ 378.66 |
| 08/82 | $ 354.84 | $ 4.14 | $ 120.98 | $ 2,281.99 | $ 379.68 |
| 09/82 | $ 236.02 | $ 2.75 | $ 119.97 | $ 2,380.19 | $ 379.27 |
| 10/82 | $ 117.80 | $ 1.37 | $ 118.96 | $ 2,477.79 | $ 377.45 |
| Interest Paid During '81-'82 is | | | | $124.28 | |
| 11/82 | $ 0.20 | $ 0.00 | $ 0.20 | $ 2,457.00 | $ 374.25 |
| Interest Paid During '81-'82 is | | | | $0.00 | |

**Listing 16-9. Fourth Interaction with LOAN2**

## 16.4 Computation of Depreciation Schedules

The final example of commercial processing involves evaluating depreciation schedules. Listing 16-10 shows the program called DEPREC that reads several input values and prints a table of output according to one of three different depreciation schedules:

- straight-line
- sum of the years
- double declining

The program also accounts for bonus depreciation during the first year, reduction in taxable income due to sales tax, and investment tax credit on new or used equipment.

Listings 16-11 through 16-15 illustrate sample interaction with DEPREC using various input parameters.

### 16.4.1 General Algorithms

DEPREC uses the following general algorithms:

- Investment Tax Credit (ITC) is assumed to be 10% of the selling price applied to the full price of new equipment, or up to $100,000 in the case of used equipment. (See the %REPLACE statement, line 11.)

- Bonus depreciation is assumed to be 10% of the selling price, up to a maximum of $2,000. (See lines 12 and 13.)

- Under all three depreciation schedules, the amount to depreciate is taken as the difference between the selling price, minus the bonus depreciation, and the residual value of the equipment.

- Under all three schedules, the depreciation value computed for the first year is prorated by month through the remainder of the fiscal year, not including bonus depreciation.

- In straight-line depreciation, the amount to depreciate is spread uniformly over the number of years in which the depreciation occurs.

- For the sum of the years, the year values are summed starting at 1, through the number of years in which depreciation takes place:

  ys = 1 + 2 + 3 + . . . + years

- The depreciation is distributed over the total number of years by computing years/ys multiplied by the depreciation value for the first year, (years-1)/ys multiplied by the remainder for the second year, and so forth, until the last year, in which 1/ys multiplied by the remaining depreciation value is taken.

- For double declining, yearly depreciation is computed as the book value divided by the number of years, which is then multiplied by 2 for new equipment, or 1.5 if the equipment is used.

DEPREC first reads the selling price, residual value, percentage sales tax, the percentage income tax bracket, the number of months remaining in the current fiscal year, and the number of years in which to depreciate the equipment. It then asks whether the equipment is new or used, and reads the depreciation schedule code for the subsequent report.

```
 1 a   /*******************************************************/
 2 a   /* This program calculates three kinds of depreciation */
 3 a   /* schedules: straight_line, sum_of_the_years, and     */
 4 a   /* double_declining.                                   */
 5 a   /*******************************************************/
 6 a   depreciate:
 7 b       procedure options(main);
 8 b       %replace
 9 b          clear_screen by '^z',
10 b          indent   by 15,
11 b          ITC_rate by .1,
12 b          bonus_rate by .1,
13 b          bonus_max  by 2000;
14 b
```

**Listing 16-10. The DEPREC Program**

```
15 b        declare
16 b           selling_price  decimal(8,2),
17 b           adjusted_price decimal(8,2),
18 b           residual_value decimal(8,2),
19 b           year_value decimal(8,2),
20 b           depreciation_value decimal(8,2),
21 b           total_depreciation decimal(8,2),
22 b           book_value decimal(8,2),
23 b           tax_rate decimal(3,2),
24 b           sales_tax decimal(8,2),
25 b           tax_bracket decimal(2),
26 b           FYD decimal(8,2),
27 b           ITC decimal(8,2),
28 b           bonus_dep decimal(8,2),
29 b           months_remaining decimal(2),
30 b           new character(4),
31 b           factor decimal(2,1),
32 b           years decimal(2),
33 b           year_sum decimal(3),
34 b           current_year decimal(2),
35 b           select_sched character(1);
36 b
37 b        declare
38 b           copy_to_list character(4),
39 b           output file variable,
40 b           (sysprint, list) file;
41 b
42 b        declare
43 b           schedules character(3) static initial ('syd'),
44 b           schedule (0:3) entry variable;
45 b
46 b        schedule (0) = error;
47 b        schedule (1) = straight_line;
48 b        schedule (2) = sum_of_years;
49 b        schedule (3) = double_declining;
50 b
51 b        open file (sysprint) stream print pagesize(0)
52 b                  title ('$con');
53 b
```

**Listing 16-10. (continued)**

```
54 c      do while('1'b);
55 c         put list(clear_screen,'^i^iDepreciation Schedule');
56 c         put skip(3) list('^i^iSelling  Price? ');
57 c         get list(selling_price);
58 c         put list('^i^iResidual Value? ');
59 c         get list(residual_value);
60 c         put list('^i^iSales Tax (%)?  ');
61 c         get list(tax_rate);
62 c         put list('^i^iTax Bracket(%)? ');
63 c         get list(tax_bracket);
64 c         put list('^i^iProRate Months? ');
65 c         get list(months_remaining);
66 c         put list('^i^iHow Many Years? ');
67 c         get list(years);
68 c         put list('^i^iNew? (yes/no)   ');
69 c         get list(new);
70 c         put edit('^i^iSchedule:',
71 c                  '^i^iStraight   (s)',
72 c                  '^i^iSum-of-Yrs (y)',
73 c                  '^i^iDouble Dec (d)? ') (a,skip);
74 c         get list(select_sched);
75 c         put list('^i^iList? (yes/no)  ');
76 c         get list(copy_to_list);
77 c         if copy_to_list = 'yes' then
78 c            open file(list) stream print title('$lst');
79 c         factor = 1.5;
80 c         if new = 'yes' then
81 c            factor = 2.0;
82 c         sales_tax = decimal(selling_price*tax_rate,12,2)/100+.005;
83 c         if new = 'yes' | selling_price <= 100000.00 then
84 c            ITC = selling_price * ITC_rate;
85 c         else
86 c            ITC = 100000 * ITC_rate;
87 c         bonus_dep = selling_price * bonus_rate;
88 c         if bonus_dep > bonus_max then
89 c            bonus_dep = bonus_max;
90 c         put list(clear_screen);
91 c         call display(sysprint);
92 c         if copy_to_list = 'yes' then
93 c            call display(list);
94 c         put skip list('^i^i^i   Type RETURN to Continue');
95 c         get skip(2);
96 c      end;
97 b
```

**Listing 16-10. (continued)**

```
 98 b   /*******************************************************/
 99 b   /* This procedure displays the various depreciation   */
100 b   /* schedules.  It calls the appropriate schedule with */
101 b   /* an index into an array of entry constants.         */
102 b   /*******************************************************/
103 b ┌─ display:
104 c │      procedure(f);
105 c │      declare
106 c │        f file;
107 c │      output = f;
108 c │      call schedule (index (schedules,select_sched));
109 c └─ end display;
110 b
111 b   /*********************************************/
112 b   /* This is a global error recovery routine. */
113 b   /*********************************************/
114 b ┌─ error:
115 c │      procedure;
116 c │      put file (output) edit('Invalid Schedule - Enter s, y, or d')
117 c │               (page,column(indent),x(8),a);
118 c │      call line();
119 c └─ end error;
120 b
121 b   /*******************************************************/
122 b   /* This procedure computes straight_line depreciation. */
123 b   /*******************************************************/
124 b ┌─ straight_line:
125 c │      procedure;
126 c │      adjusted_price = selling_price - bonus_dep;
127 c │      put file (output) edit('S T R A I G H T   L I N E')
128 c │               (page,column(indent),x(14),a);
129 c │      call header();
130 c │      depreciation_value = adjusted_price - residual_value;
131 c │      book_value = adjusted_price;
132 c │      total_depreciation = 0;
```

**Listing 16-10. (continued)**

```
133 d     do current_year = 1 to years;
134 d        year_value = decimal(depreciation_value/years,8,2) + .005;
135 d        if current_year = 1 then
136 e        do;
137 e           year_value = year_value * months_remaining / 12;
138 e           FYD = year_value;
139 e        end;
140 d        depreciation_value = depreciation_value - year_value;
141 d        total_depreciation = total_depreciation + year_value;
142 d        book_value = adjusted_price - total_depreciation;
143 d        call print_line();
144 d     end;
145 c     call summary();
146 c  end straight_line;
147 b
148 b  /*************************************************/
149 b  /* This procedure computes depreciation based on */
150 b  /* the sum_of_the_years.                         */
151 b  /*************************************************/
152 b  sum_of_years:
153 c     procedure;
154 c     adjusted_price = selling_price - bonus_dep;
155 c     put file (output) edit('S U M   O F   T H E   Y E A R S')
156 c              (page,column(indent),x(11),a);
157 c     call header();
158 c     depreciation_value = adjusted_price - residual_value;
159 c     book_value = adjusted_price;
160 c     total_depreciation = 0;
161 c     year_sum = 0;
162 d     do current_year = 1 to years;
163 d        year_sum = year_sum + current_year;
164 d     end;
```

**Listing 16-10. (continued)**

```
165 c
166 d     do current_year = 1 to years;
167 d        year_value = decimal(depreciation_value *
168 d        (years - current_year + 1),12,2)/ year_sum + .005;
169 d        if current_year = 1 then
170 e           do;
171 e              year_value = year_value * months_remaining / 12;
172 e              FYD = year_value;
173 e           end;
174 d        depreciation_value = depreciation_value - year_value;
175 d        total_depreciation = total_depreciation + year_value;
176 d        book_value = adjusted_price - total_depreciation;
177 d        call print_line();
178 d     end;
179 c     call summary();
180 c  end sum_of_years;
181 b
182 b  /*******************************************/
183 b  /* This procedure computes double_declining */
184 b  /* depreciation.                           */
185 b  /*******************************************/
186 b  double_declining:
187 c     procedure;
188 c     adjusted_price = selling_price - bonus_dep;
189 c     put file (output) edit('D O U B L E   D E C L I N I N G')
190 c              (page,column(indent),x(10),a);
191 c     call header();
192 c     depreciation_value = adjusted_price - residual_value;
193 c     book_value = adjusted_price;
194 c     total_depreciation = 0;
195 d     do current_year = 1 to years
196 d        while (depreciation_value > 0);
197 d        year_value = decimal(book_value/years,8,2) * factor+.005;
198 d        if current_year = 1 then
199 e           do;
200 e              year_value = year_value * months_remaining / 12;
201 e              FYD = year_value;
202 e           end;
203 d        if year_value > depreciation_value then
204 d           year_value = depreciation_value;
205 d        depreciation_value = depreciation_value - year_value;
206 d        total_depreciation = total_depreciation + year_value;
207 d        book_value = adjusted_price - total_depreciation;
208 d        call print_line();
209 d     end;
210 c     call summary();
211 c  end double_declining;
212 b
```

**Listing 16-10. (continued)**

```
213 b   /************************************************/
214 b   /* This procedure prints an output header record. */
215 b   /************************************************/
216 b ┌ header:
217 c │     procedure;
218 c │     declare
219 c │        new_or_used character(5);
220 c │
221 c │   ┌ if new = 'yes' then
222 c │   │    new_or_used = ' New';
223 c │   └ else
224 c │        new_or_used = ' Used';
225 c │     put file (output) edit(
226 c │     '-------------------------------------------------',
227 c │     '¦',selling_price+sales_tax,new_or_used,
228 c │      residual_value,' Residual Value¦',
229 c │     '¦',months_remaining,' Months Left ',
230 c │         tax_rate,'% Tax',tax_bracket,'% Tax Bracket¦')
231 c │     (2(skip,column(indent),a),
232 c │     2(p'B$$,$$$,$$9.V99',a),
233 c │     skip,column(indent),a,x(5),f(2),a,2(x(2),p'B99',a));
234 c │
235 c │     put file (output) edit(
236 c │     '-------------------------------------------------',
237 c │     '¦ Y ¦ Depreciation ¦ Depreciation ¦  Book Value  ¦',
238 c │     '¦ r ¦   For Year   ¦  Remaining   ¦              ¦',
239 c │     '-------------------------------------------------')
240 c │     (skip,column(indent),a);
241 c └ end header;
242 b
243 b   /*****************************************/
244 b   /* This procedure prints the current line. */
245 b   /*****************************************/
246 b ┌ print_line:
247 c │     procedure;
248 c │     put file (output) edit(
249 c │         '¦',current_year,
250 c │         ' ¦',year_value,
251 c │         ' ¦',depreciation_value,
252 c │         ' ¦',book_value,' ¦')
253 c │         (skip,column(indent),a,f(2),4(a,p'$z,zzz,zz9v.99'));
254 c └end print_line;
255 b
```

**Listing 16-10. (continued)**

```
256 b   /************************************************/
257 b   /* This procedure prints the summary of values for */
258 b   /* each type of depreciation schedule.            */
259 b   /************************************************/
260 b ┌─summary:
261 c │     procedure;
262 c │     declare
263 c │        adj_ITC decimal(8,2),
264 c │        total decimal(8,2),
265 c │        direct decimal(8,2);
266 c │
267 c │     call line();
268 c │     adj_ITC = ITC * 100 / tax_bracket;
269 c │     total = FYD + sales_tax + adj_ITC + bonus_dep;
270 c │     direct = total * tax_bracket / 100;
271 c │     put file (output) edit(
272 c │     '¦      First Year Reduction in Taxable Income    ¦',
273 c │     '-----------------------------------------------',
274 c │     '¦      Depreciation              ',FYD,        '¦',
275 c │     '¦      Sales Tax                 ',sales_tax,  '¦',
276 c │     '¦      ITC (Adjusted)            ',adj_ITC,    '¦',
277 c │     '¦      Bonus Depreciation        ',bonus_dep,  '¦',
278 c │     '¦                                ------------  ¦',
279 c │     '¦      Total for First Year      ',total,      '¦',
280 c │     '¦      Direct Reduction in Tax   ',direct,     '¦')
281 c │     (2(skip,column(indent),a),2(4(skip,column(indent),a,
282 c │     p'$z,zzz,zz9v.99',x(3),a),skip,column(indent),a));
283 c │     call line();
284 c └─end summary;
285 b
286 b   /*****************************************/
287 b   /* This procedure prints a line of dashes. */
288 b   /*****************************************/
289 b ┌─line:
290 c │     procedure;
291 c │     put file (output) edit(
292 c │     '-----------------------------------------------')
293 c │     (skip,column(indent),a);
294 c └─end line;
295 b
296 b
297 b   end depreciate;
```

**Listing 16-10. (continued)**

### 16.4.2 Selecting the Schedule

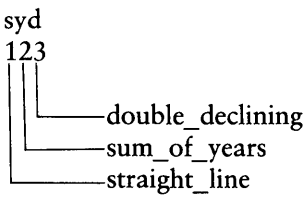
There are two constructs in DEPREC that merit special consideration.

DEPREC uses an array of ENTRY variables to select one of three schedules. Line 42 defines the array with a subscript range of zero to three. Lines 46 to 49 initialize the individual elements of the array, and allow indirect calls to either the ERROR subroutine or one of the depreciation schedule handling subroutines. The actual calls to the subroutines occur later in the program.

The schedule selection takes place on line 74, where DEPREC reads one of the characters s, y, or d from the console into the character variable select_sched. Line 93 then invokes the DISPLAY subroutine which performs the actual dispatch to the schedule handler with the statement on line 108:

```
call schedule (index (schedules,select_sched));
```

This particular statement works as follows. Line 43 defines the variable schedules, and initializes them to the character string 'syd', where each letter corresponds to one of the schedule-handling following subroutines:




Therefore, the statement

```
call schedule (index(schedules,select_sched));
```

is equivalent to,

```
call schedule (index(syd,select_sched));
```

and for the valid inputs s, y, or d, produces 1, 2, or 3 respectively.

Thus, if select_sched is s, the call statement evaluates to,

```
call schedule(1);
```

which calls the subroutine STRAIGHT_LINE. Similarly, an input of y or d evaluates to,

```
call schedule(2);  or  call schedule(3);
```

producing a call to SUM_OF_YEARS or DOUBLE_DECLINING respectively.

If the value of select_sched is not s, y, or d, then the INDEX function returns a zero value. All invalid character input values produce,

```
call schedule(0);
```

which calls the ERROR subroutine and prints the error message.

### 16.4.3 Displaying the Output

Another construct of DEPREC is the output file variable, defined on line 39. During the parameter input phase, DEPREC prompts you with:

```
List? (yes/no)
```

A yes response sends the output from the program to both the console and the list device.

Line 40 declares two file constants, sysprint and list, to address the console and the list device. DEPREC first opens the console file, line 51, using an infinite page length to avoid form-feed characters.

On any iteration of the main DO-group, if you give an affirmative response on line 77, DEPREC subsequently opens the list device, line 78. This statement can be executed several times during a particular execution of the program, but only the first OPEN statement has any effect; PL/I ignores the OPEN statement if the file is already open.

Line 91 calls the DISPLAY subroutine to compute and display the output report for a specific set of input values. DISPLAY has a single actual parameter consisting of the file constant sysprint that is defined as the formal parameter f on line 104. Line 107 assigns the formal parameter to the global variable output. Subsequent PUT statements write data to the console, producing the first report.

On line 92, if the variable copy_to_list has the character value yes, then DEPREC calls DISPLAY once again. This time, the actual parameter is list, corresponding to the system list device. Thus, the output file variable is indirectly assigned the value list, and all PUT statements that reference file output send data to the printer. This results in both a soft and hard copy of the report.

DEPREC uses several different forms of decimal arithmetic. Examine the various declarations while cross-checking the output formats with the displayed results.

```
A>deprec

                        Depreciation Schedule

Selling  Price?  200000
Residual Value?  40000
Sales Tax (%)?   6
Tax Bracket(%)?  50
ProRate Months?  10
How Many Years?  7
New? (yes/no)    no
Schedule:
Straight   (s)
Sum-of-Yrs (y)
Double Dec (d)?  d
List? (yes/no)   no

                      D O U B L E    D E C L I N I N G
-------------------------------------------------
|   $212,000.00 Used     $40,000.00 Residual Value|
|     10 Months Left     06% Tax   50% Tax Bracket|
-------------------------------------------------
| Y | Depreciation | Depreciation |  Book Value  |
| r |   For Year   |  Remaining   |              |
-------------------------------------------------
| 1 |$   35,357.14 |$  122,642.86 |$  162,642.86 |
| 2 |$   34,852.04 |$   87,790.82 |$  127,790.82 |
| 3 |$   27,383.75 |$   60,407.07 |$  100,407.07 |
| 4 |$   21,515.79 |$   38,891.28 |$   78,891.28 |
| 5 |$   16,905.27 |$   21,986.01 |$   61,986.01 |
| 6 |$   13,282.71 |$    8,703.30 |$   48,703.30 |
| 7 |$    8,703.30 |$        0.00 |$   40,000.00 |
-------------------------------------------------
|      First Year Reduction in Taxable Income    |
-------------------------------------------------
|      Depreciation              $   35,357.14  |
|      Sales Tax                 $   12,000.00  |
|      ITC (Adjusted)            $   20,000.00  |
|      Bonus Depreciation        $    2,000.00  |
|                                ------------   |
|      Total for First Year      $   69,357.14  |
|      Direct Reduction in Tax   $   34,678.57  |
-------------------------------------------------
```

**Listing 16-11. First Interaction with DEPREC**

```
                        Depreciation Schedule

Selling Price?  ,
Residual Value?  ,
Sales Tax (%)?  ,
Tax Bracket(%)?  ,
ProRate Months?  8
How Many Years?  ,
New? (yes/no)    yes
Schedule:
Straight   (s)
Sum-of-Yrs (y)
Double Dec (d)?  y
List? (yes/no)   no


          S U M   O F   T H E   Y E A R S
-------------------------------------------------
|   $212,000.00 New      $40,000.00 Residual Value|
|      8 Months Left     06% Tax   50% Tax Bracket|
-------------------------------------------------
| Y | Depreciation | Depreciation |  Book Value  |
| r |   For Year   |  Remaining   |              |
-------------------------------------------------
| 1 |$   26,333.33 |$  131,666.67 |$  171,666.67 |
| 2 |$   28,214.29 |$  103,452.38 |$  143,452.38 |
| 3 |$   18,473.64 |$   84,978.74 |$  124,978.74 |
| 4 |$   12,139.82 |$   72,838.92 |$  112,838.92 |
| 5 |$    7,804.17 |$   65,034.75 |$  105,034.75 |
| 6 |$    4,645.34 |$   60,389.41 |$  100,389.41 |
| 7 |$    2,156.76 |$   58,232.65 |$   98,232.65 |
-------------------------------------------------
|      First Year Reduction in Taxable Income    |
-------------------------------------------------
|      Depreciation              $   26,333.33  |
|      Sales Tax                 $   12,000.00  |
|      ITC (Adjusted)            $   40,000.00  |
|      Bonus Depreciation        $    2,000.00  |
|                                ------------   |
|      Total for First Year      $   80,333.33  |
|      Direct Reduction in Tax   $   40,166.66  |
-------------------------------------------------
```

**Listing 16-12. Second Interaction with DEPREC**

```
                    Depreciation Schedule

Selling Price? 310000
Residual Value? 30000
Sales Tax (%)? ,
Tax Bracket(%)? ,
ProRate Months? 12
How Many Years? 5
New? (yes/no)  yes
Schedule:
Straight   (s)
Sum-of-Yrs (y)
Double Dec (d)? d
List? (yes/no)  no

          D O U B L E   D E C L I N I N G
-------------------------------------------------
|   $328,600.00 New      $30,000.00 Residual Value|
|     12 Months Left     06% Tax   50% Tax Bracket|
-------------------------------------------------
| Y | Depreciation | Depreciation |  Book Value  |
| r |   For Year   |  Remaining   |              |
-------------------------------------------------
| 1 |$  123,200.00 |$  154,800.00 |$  184,800.00 |
| 2 |$   73,920.00 |$   80,880.00 |$  110,880.00 |
| 3 |$   44,352.00 |$   36,528.00 |$   66,528.00 |
| 4 |$   26,611.20 |$    9,916.80 |$   39,916.80 |
| 5 |$    9,916.80 |$        0.00 |$   30,000.00 |
-------------------------------------------------
|      First Year Reduction in Taxable Income    |
-------------------------------------------------
|      Depreciation              $  123,200.00  |
|      Sales Tax                 $   18,600.00  |
|      ITC (Adjusted)            $   62,000.00  |
|      Bonus Depreciation        $    2,000.00  |
|                                ------------   |
|      Total for First Year      $  205,800.00  |
|      Direct Reduction in Tax   $  102,900.00  |
-------------------------------------------------
```

**Listing 16-13. Third Interaction with DEPREC**

```
                    551    Depreciation Schedule

Selling Price?  ,
Residual Value?  ,
Sales Tax (%)?   ,
Tax Bracket(%)?  ,
ProRate Months?  ,
How Many Years?  ,
New? (yes/no)    ,
Schedule:
Straight   (s)
Sum-of-Yrs (y)
Double Dec (d)?  s
List? (yes/no)   ,


                  S T R A I G H T   L I N E
-------------------------------------------------
|   $328,600.00 New      $30,000.00 Residual Value|
|     12 Months Left     06% Tax   50% Tax Bracket|
-------------------------------------------------
| Y | Depreciation | Depreciation |  Book Value  |
| r |   For Year   |  Remaining   |              |
-------------------------------------------------
| 1 |$   55,600.00 |$  222,400.00 |$  252,400.00 |
| 2 |$   44,480.00 |$  177,920.00 |$  207,920.00 |
| 3 |$   35,584.00 |$  142,336.00 |$  172,336.00 |
| 4 |$   28,467.20 |$  113,868.80 |$  143,868.80 |
| 5 |$   22,773.76 |$   91,095.04 |$  121,095.04 |
-------------------------------------------------
|      First Year Reduction in Taxable Income    |
-------------------------------------------------
|      Depreciation              $   55,600.00  |
|      Sales Tax                 $   18,600.00  |
|      ITC (Adjusted)            $   62,000.00  |
|      Bonus Depreciation        $    2,000.00  |
|                                ------------   |
|      Total for First Year      $  138,200.00  |
|      Direct Reduction in Tax   $   69,100.00  |
-------------------------------------------------
```

**Listing 16-14. Fourth Interaction with DEPREC**

*End of Section 16*

References: Sections 3.1, 3.5, 4.2, 11.3 LRM

# Section 17
# Internal Data Representation

This section describes how PL/I represents data internally. This knowledge is vital when using based variables to overlay storage so you do not destroy adjacent storage locations. Knowledge of the internal data representation is also useful when so you want to interface assembly language routines with high-level language programs and the PL/I Run-time Subroutine Library.

**Note:** the discussion in this section applies to the implementation of PL/I for both 8-bit and 16-bit processors.

## 17.1 FIXED BINARY Representation

PL/I stores FIXED BINARY data in one of two forms, depending upon the declared precision. It stores FIXED BINARY values with precision 1-7 in single-byte locations, and values with precision 8-15 in a word (double-byte) location. With multibyte storage, PL/I stores the least significant byte first.

PL/I represents all FIXED BINARY data in two's complement form, allowing single-byte values in the range -128 to +127, and word values in the range -32768 to +32767.

The following figure shows the representation of storage in both single-byte and double-byte locations for the values 0, 1, and -1. Each boxed value represents a byte of memory, and is shown in both binary and hexadecimal values.

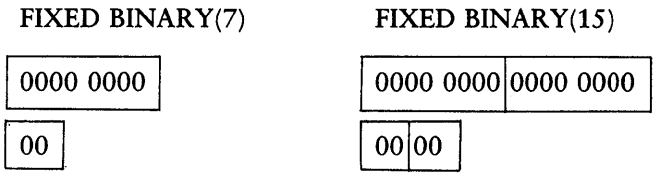
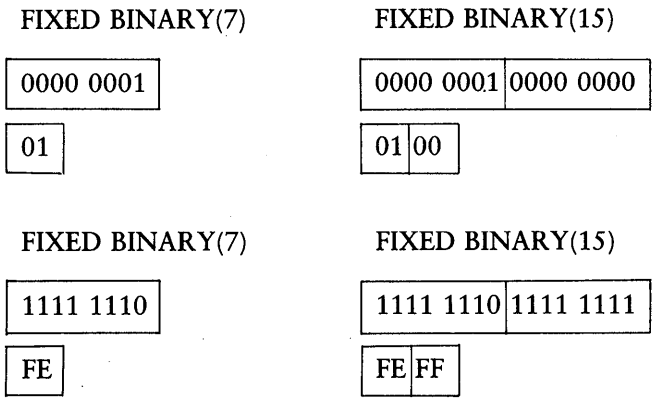



**Figure 17-1. FIXED BINARY Representation**




## 17.2 FLOAT BINARY Representation

PL/I stores single-precision floating-point binary numbers in four consecutive bytes. The 32 bits contain the following fields: a 23-bit mantissa, a sign bit, and an 8-bit exponent. The least significant byte of the mantissa appears first in memory.

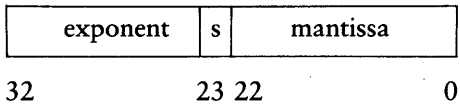
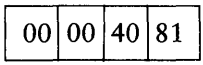



**Figure 17-2. Single-precision Floating-point Binary**

PL/I normalizes floating-point numbers so the most significant bit of the mantissa is always 1 for nonzero numbers. Because the most significant bit of the mantissa must be 1 for nonzero numbers, PL/I replaces this bit position with the sign. PL/I represents a zero mantissa with an exponent byte of 00.

In order to make certain kinds of comparisons easier, the binary exponent byte has a bias of 80 (hexadecimal), so that 81 represents an exponent of 1 while 7F represents an exponent of -1.

Suppose a floating-point binary value appears in memory as shown in the following example:




In this case, the mantissa is a bit stream of the form

```
  4    0

0100 0000 . . .
```

and the high-order bit equal to zero indicates that the mantissa sign is positive. Normalizing the number produces the bit stream:

```
1100 0000 . . .
```

The exponent 81 has a bias of 80, so the binary exponent is 1. This means that the binary point is one position to the right, resulting in the binary value

```
1 100 0000 . . .
^
```

1^1 in binary represents $2^0 \hat{} 2^{-1}$; therefore 1^1 base 2 is equivalent to 1.5 base 10.

```
00 00 40 81
```

is the floating-point binary representation of the decimal number 1.5.

PL/I stores double-precision floating-point binary numbers in eight consecutive bytes. The 64 bits contain the following fields: a 52-bit mantissa, an 11-bit exponent with a bias of 3FF(hexadecimal), and a sign-bit.

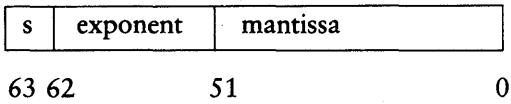
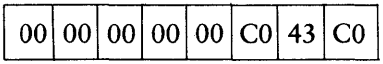



**Figure 17-3. Double-precision Floating-Point Binary**

For example, suppose that a floating-point binary value appears in memory as shown in the following:




In this case, the mantissa is a bit stream of the form,

```
3     C     0

0011  1100  0000 . . .
```

Normalizing the number produces,

```
1001  1110  0000 . . .
```

The exponent evaluates as follows:

```
C     0     4

1100  0000  0100
```

The high-order bit is 1 so the sign is negative. Ignoring the sign bit yields an exponent of,

```
  4     0     4
0100  0000  0100
```

which has a bias of 3FF, so the real exponent is,

```
  404
-3FF
    5
```

Therefore, the binary number is,

```
1001 11 10 0000 . . .
     ^
```

which is 39.5 in decimal. Thus, the eight-byte value,

```
00 00 00 00 00 C0 43 C0
```

is the double-precision float-binary representation of the decimal number -39.5.

## 17.3 FIXED DECIMAL Representation

PL/I stores FIXED DECIMAL data items in packed BCD (Binary Coded Decimal) form. Each BCD digit occupies a half-byte, or nibble. PL/I stores the least significant BCD pair first, with one BCD digit position reserved for the sign. Positive numbers have a 0 sign, while negative numbers have a 9 in the high-order sign digit position.

The number of bytes occupied by a FIXED DECIMAL number depends upon its declared precision. Given a decimal number with precision p, PL/I reserves a number of bytes equal to:

```
FLOOR((p + 2)/2)
```

where p varies between 1 and 15. This results in a minimum of 1 byte and a maximum of 8 bytes to hold a FIXED DECIMAL data item.

For example, if you declare the number 12345 with precision 5, then PL/I reserves FLOOR((5 + 2)/2) = 3 bytes of storage and represents the number as:

```
45 23 01
```

PL/I stores negative FIXED DECIMAL numbers in ten's complement form. To derive the ten's complement of a number, first derive the nine's complement and then add 1 to the result. For example, the number -2 expressed in ten's complement is,

(9 - 2) + 1 = 8

Adding the sign digit gives,

98

If you declare -2 with precision 5, then PL/I represents it as:

98 99 99

## 17.4 CHARACTER Representation

PL/I stores character data in one of two forms, depending upon the declaration. It stores fixed-length character strings, declared as CHARACTER(n) in n contiguous bytes, with the first character in the string stored lowest in memory.

PL/I reserves n + 1 bytes for variables declared as CHARACTER(n) VARYING with the extra byte holding the character string's length, ranging from 0 to 254. The maximum length of either type of string is 254 characters.

As an example, suppose the variable A is declared as CHARACTER(20). The assignment

```
A = 'Walla Walla Wash';
```

results in the following storage allocation,

W a l l a b W a l l a b W a s h x x x x

where b represents a blank, and x represents an undefined character position. If A is declared as CHARACTER(20) VARYING data, PL/I stores the same string as

10 W a l l a b W a l l a b W a s h x x x x

where 10 is the (hexadecimal) string length.

## 17.5 BIT Representation

PL/I represents bit-string data in two forms, depending upon the declared precision. It stores bit strings of length 1-8 in a single byte, and bit strings of length 9-16 in a word (double-byte) value. PL/I stores the least significant byte of a word value first in memory. Bit values are stored left-justified, and if the precision is not exactly 8 or 16 bits, the bits to the right are ignored.

The following figure shows the storage for the bit-string constant values '1'b, 'A0'b4, and '1234'b4 in both single- and double-byte locations. Each boxed value represents a byte.

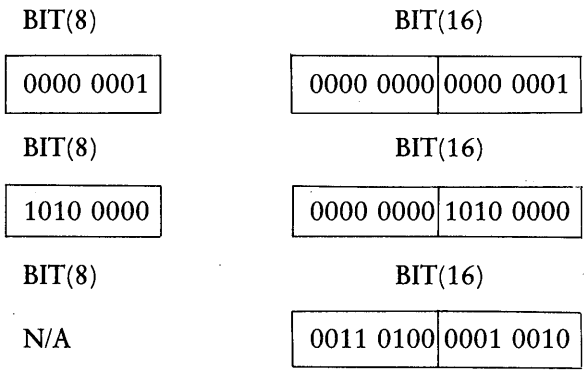



**Figure 17-4. Bit-string Data Representation**

## 17.6 POINTER, ENTRY and LABEL Data

PL/I stores variables that provide access to memory addresses as two contiguous bytes, with the low-order byte stored first. POINTER, ENTRY, and LABEL data items appear as

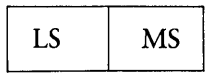



**Figure 17-5. POINTER, ENTRY, and LABEL Data Storage**

where LS denotes the least significant half of the address, and MS denotes the most significant portion. MS contains the page address, where each memory page is 256 bytes, and LS contains the offset within the page.

## 17.7 File Constant Representation

PL/I associates each file constant with a File Parameter Block (FPB). The FPB occupies 57 contiguous bytes containing various fields, some of which are implementation dependent.

**Note:** each file declaration causes a static allocation for the associated FPB. When you open the file, there is an additional overhead of 50 bytes, including the operating system's FCB and the amount specified for buffer space. The run-time system dynamically allocates this storage from the free storage area.

## 17.8 Aggregate Storage

PL/I stores aggregate data items contiguously with no filler bytes. Bit data is always stored unaligned. Arrays are stored in row-major order, with the rightmost subscript varying fastest.

For example, the declaration

```
declare A(2,2,2);
```

results in the following storage allocation:

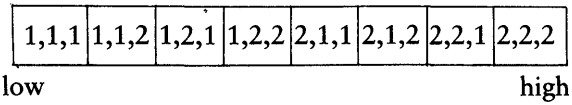



**Figure 17-6. Aggregate Storage**

*End of Section 17*

# Section 18
# Interface Conventions

This section describes a standard set of conventions for interfacing PL/I programs with assembly language routines and with programs written in other high-level languages. This section also describes the mechanism for making direct operating system calls using a set of optional subroutines not included in the Run-time Subroutine Library.

## 18.1 Parameter Passing Conventions

You can pass parameters between a PL/I program and an assembly language routine by loading a register pair with the address of a Parameter Block containing pointer values. These pointers in turn lead to the actual parameter values. The number of parameters and the parameter length and type must be determined implicitly by agreement between the calling program and called subroutine. The following figure illustrates the concept. The address fields are arbitrary.

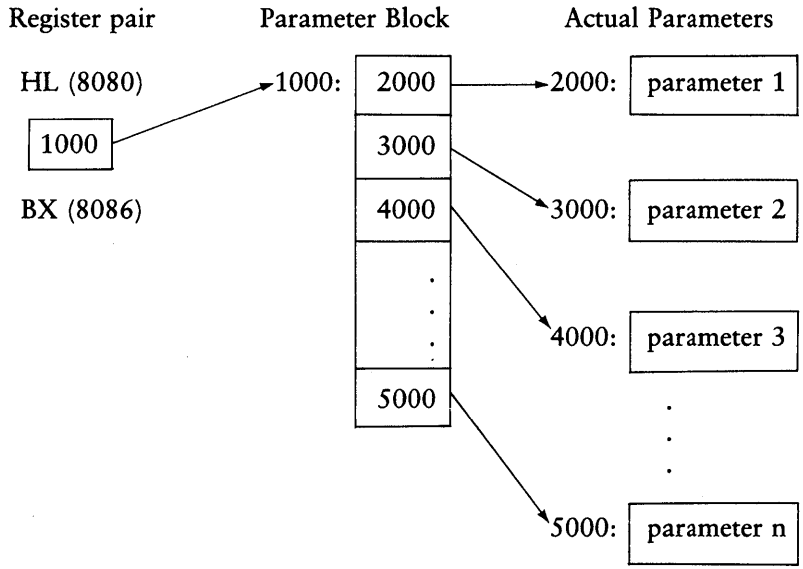



**Figure 18-1. PL/I Parameter Passing Mechanism**

The following example illustrates this parameter passing mechanism. Suppose a PL/I program uses a considerable number of floating-point divide operations, where each division is by a power of two. Suppose also that the iterative loop where the divisions occur is speed-critical, and that it is useful to have an assembly language subroutine to perform the division.

The assembly language routine simply decreases the binary exponent of the floating-point number for each power of two in the division. Decreasing the exponent effectively performs the divide operation without the overhead involved in unpacking the number, performing the general division operation, and repacking the result. During the division, the assembly language routine can produce underflow, and must signal the UNDERFLOW condition to the PL/I program if this occurs.

The following three listings show programs that demonstrate parameter passing. Listing 18-1 shows the program DTEST, which tests the division operation. Listing 18-2 shows DIV2.ASM, the 8080 assembly language subroutine that performs the division. On line 8, DTEST defines DIV2 as an external entry constant with two parameters: a FIXED(7) and a floating-point binary value. Listing 18-3 shows DIV2.A86, which is the same subroutine in 8086 assembly language.

On each iteration of the DO-group, DTEST stores the test value 100 into f (line 13), and passes it to the DIV2 subroutine (line 14). At each call to DIV2, DTEST changes the value of f to f/(2**i) and prints it using a PUT statement. At the point of call, DIV2 receives two addresses that correspond to the two parameters i and f.

Upon entry, DIV2 loads the value of i to the accumulator, and sets the appropriate register pair to point to the exponent field of the input floating-point number. If the exponent is zero, DIV2 returns immediately, because the resulting value is zero.

Otherwise, the subroutine loops at the label dby2 while counting down the exponent as the power of two diminishes to zero. If the exponent reaches zero during this counting process, DIV2 signals the UNDERFLOW condition.

In DIV2, the call to ?signal demonstrates the assembly language format for parameters that use the interface. The ?signal subroutine is part of the PL/I Run-time Subroutine Library (PLILIB.IRL).

This subroutine loads the appropriate register pair with the address of the Signal Parameter List, denoted by siglst. The Signal Parameter List, in turn, is a Parameter Block of four addresses leading to the signal code sigcode, the signal subcode sigsub, the filename indicator sigfil (not used here), and the auxiliary message sigaux that is the last parameter.

The auxiliary message can provide additional information when an error occurs. The signal subroutine prints the message until it either exhausts the string length (32, in this case), or it encounters a binary 00 in the string.

Listing 18-4 shows the abbreviated output from this test program. The loop counter i becomes negative when it reaches 128, but the DIV2 subroutine treats this value as an unsigned magnitude value; thus UNDERFLOW occurs when i reaches −123.

```
 1 a   /*****************************************************/
 2 a   /* This program tests an assembly language routine to */
 3 a   /* do floating-point division.                        */
 4 a   /*****************************************************/
 5 a┌──dtest:
 6 b│       procedure options(main);
 7 b│       declare
 8 b│         div2 entry(fixed(7),float),
 9 b│         i fixed(7),
10 b│         f float;
11 b│
12 c│     ┌──do i = 0 by 1;
13 c│     │    f = 100;
14 c│     │    call div2(i,f);
15 c│     │    put skip list('100 / 2 **',i,'=',f);
16 c│     └──end;
17 b│
18 b└──end dtest;
```

**Listing 18-1. The DTEST Program**

```
        title   'division by power of two'
        public  div2
        extrn   ?signal
;       entry:
;               p1 -> fixed(7) power of two
;               p2 -> floating-point number
;       exit:
;               p1 -> (unchanged)
;               p2 -> p2 / (2**p1)
div2:                   ;HL = .low(.p1)
        mov     e,m     ;low(.p1)
        inx     h       ;HL = .high(.p1)
        mov     d,m     ;DE = .p1
        inx     h       ;HL = .low(p2)
        ldax    d       ;a = p1 (power of two)
        mov     e,m     ;low(.p2)
        inx     h       ;HL = .high(.p2)
        mov     d,m     ;DE = .p2
        xchg            ;HL = .p2
;
;       A = power of 2, HL = .low byte of fp num
        inx     h       ;to middle of mantissa
        inx     h       ;to high byte of mantissa
        inx     h       ;to exponent byte
        inr     m
        dcr     m       ;p2 already zero?
        rz              ;return if so
dby2:   ;divide by two
        ora     a       ;counted power of 2 to zero?
        rz              ;return if so
        dcr     a       ;count power of two down
        dcr     m       ;count exponent down
        jnz     dby2    ;loop again if no underflow
```

**Listing 18-2. DIV2.ASM Assembly Language Program (8080)**

```
;underflow occurred, signal underflow condition
        lxi     h,siglst;signal parameter list
        call    ?signal ;signal underflow
        ret             ;normally, no return
;
        dseg
siglst: dw      sigcod  ;address of signal code
        dw      sigsub  ;address of subcode
        dw      sigfil  ;address of file code
        dw      sigaux  ;address of aux message
;       end of parameter vector, start of params
sigcod: db      3       ;03 = underflow
sigsub: db      128     ;arbitrary subcode for id
sigfil: dw      0000    ;no associated file name
sigaux: dw      undmsg  ;0000 if no aux message
undmsg: db      32,'Underflow in Divide by Two',0
        end
```

**Listing 18-2. (continued)**

```
;       Routine to divide single precision float value by 2

        cseg
        public  div2
        extrn   ?signal:near

;       entry:
;               p1 -> fixed(7) power of two
;               p2 -> floating point number
;       exit:
;               p1 -> (unchanged)
;               p2 -> p2 / (2**p1)

div2:                   ;BX = .low(.p1)
        mov     si,[bx]         ;SI = .p1
        mov     bx,2[bx]        ;BX = .p2
        lods    al              ;AL = p1 (power of 2)

;       AL = power of 2, BX = .low byte of fp num

        cmp     byte ptr 3[bx],0  ;p2 already zero?
        jz      done            ;exit if so
```

**Listing 18-3. DIV2.A86 Assembly Language Program (8086)**

```
dby2:                   ;divide by two
        or      al,al           ;counted power of 2 to zero?
        jz      done            ;return if so
        dec     al              ;count power of two down
        dec     byte ptr 3[bx]  ;count exponent down
        jnz     dby2            ;loop again if no underflow

;       Underflow occurred, signal underflow condition

        mov     bx,offset siglst;signal parameter list
        call    ?signal         ;signal underflow
done:   ret                     ;normally, no return

        dseg
siglst  dw      offset sigcod   ;address of signal code
        dw      offset sigsub   ;address of subcode
        dw      offset sigfil   ;address of file code
        dw      offset sigaux   ;address of aux message
;       end of parameter vector, start of params
sigcod  db      3               ;03 = underflow
sigsub  db      128             ;arbitrary subcode for id
sigfil  dw      0000            ;no associated file name
sigaux  dw      offset undmsg   ;0000 if no aux message
undmsg  db      32,'Underflow in Divide by Two',0

        end
```

**Listing 18-3. (continued)**

```
A>dtest

100 / 2 **       0 = 1.000000E+02
100 / 2 **       1 = 5.000000E+01
100 / 2 **       2 = 2.500000E+01
100 / 2 **       3 = 1.250000E+01
100 / 2 **       4 = 0.625000E+01
100 / 2 **       5 = 3.125000E+00
100 / 2 **       6 = 1.562500E+00
100 / 2 **       7 = 0.781250E+00
100 / 2 **       8 = 3.906250E-01
100 / 2 **       9 = 1.953125E-01
100 / 2 **      10 = 0.976562E-01
    .
    .
    .
    .
    .
100 / 2 **     127 = 0.587747E-36
100 / 2 **    -128 = 2.938735E-37
100 / 2 **    -127 = 1.469367E-37
100 / 2 **    -126 = 0.734683E-37
100 / 2 **    -125 = 3.673419E-38
100 / 2 **    -124 = 1.836709E-38
100 / 2 **    -123 = 0.918354E-38
100 / 2 **    -122 = 4.591774E-39
UNDERFLOW (128), Underflow in Divide By Two
Traceback: 017F 011B
A>
```

**Listing 18-4. DTEST Output (abbreviated)**

## 18.2 Returning Values from Functions

As an alternative to returning values through a Parameter Block, PL/I has subroutines that produce function values that are then returned directly in the registers or on the stack. This section shows the conventions for returning data as functional values. References to 8086 registers are in parentheses.

### 18.2.1 Returning FIXED BINARY Data

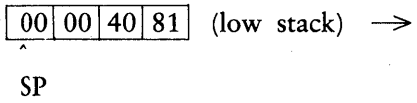
Functions that return FIXED BINARY data items do so by leaving the result in a register, or register pair, depending upon the precision of the data item.

PL/I returns FIXED BINARY data with precision 1-7 in the A(AL) register, and data with precision 8-15 in the HL(BX) register pair. It is always safe to return the value in HL(BX), and copy the low-order byte to A(AL) so register A(AL) is equal to register L(BL) upon return.

### 18.2.2 Returning FLOAT BINARY Data

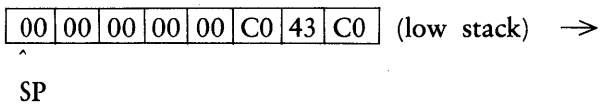
PL/I returns single-precision floating-point numbers as four contiguous bytes on the stack. The low-order byte of the mantissa is at the top of the stack, followed by the middle byte, then the high byte. The fourth byte is the exponent of the number.

For example, PL/I returns the value 1.5 as:




PL/I returns double-precision floating-point numbers as eight contiguous bytes on the stack. The low-order byte of the mantissa is at the top of the stack. The exponent occupies three nibbles: the eighth byte, and the high-order nibble of the seventh byte.

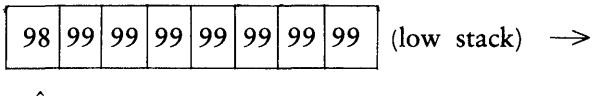
For example, PL/I returns the value −39.5 as:



### 18.2.3 Returning FIXED DECIMAL Data

PL/I returns FIXED DECIMAL data as 8 contiguous bytes on the stack. The low-order BCD pair is at the top of the stack. The number is represented in nine's complement form, and sign-extended through the high-order digit position, with a positive sign denoted by 0, and a negative sign denoted by 9.

For example, PL/I returns the decimal number −2 as:

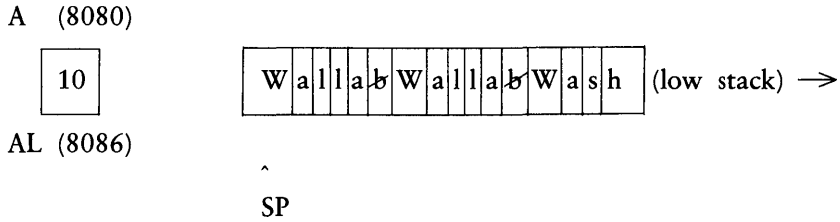



### 18.2.4 Returning CHARACTER Data

PL/I returns CHARACTER data items on the stack, with the length of the string in a register. For example, PL/I returns the string

```
'Walla Walla Wash'
```

as shown in the following diagram:




where register contains the string length 10 (hexadecimal), and the Stack Pointer SP addresses the first character in the string.

### 18.2.5 Returning BIT Data

PL/I returns bit-string data in a register, or register pair, depending upon the precision of the data item.

PL/I returns bit strings of length 1-8 in the A(AL) register, and bit strings of length 9-16 in the HL(BX) register pair. Bit strings are left justified in their fields, so the BIT(1)

value true is returned in the HL(BX) register as 80 (hexadecimal). It is safe to return a bit value in the HL(BX) register pair and copy the high-order byte in A(AL), so register A(AL) is equal to register H(BH) upon return.

### 18.2.6 Returning POINTER, ENTRY, and LABEL Variables

PL/I returns POINTER, ENTRY, and LABEL variables in the HL(BX) register pair. When returning a label variable that can be the target of a GOTO operation, the subroutine containing the label must restore the stack to the proper level when control reaches the label.

The following program listings illustrate the concept of returning a functional value. Listing 18-5 shows the program called FDTEST that is similar to the previous floating-point divide test. However, FDTEST includes an entry definition for an assembly language subroutine called FDIV2 that returns the result on the stack. Listing 18-6 shows FDIV2.ASM in 8080 assembly language, and Listing 18-7 shows FDIV2.A86, the same routine in 8086 assembly language.

FDIV2 resembles the previous subroutine DIV2 with some minor changes. First, FDIV2 loads the input floating-point value into the BC(CX) and DE(DX) registers so that it can manipulate a temporary copy and not affect the original input value FDIV2 then decreases the exponent field in register B(CH) by the input count, and returns it on the stack before executing the PCHL instruction.

```
 1 a   /*****************************************************/
 2 a   /* This program tests the assembly language routine */
 3 a   /* called FDIV2 which returns a FLOAT BINARY value. */
 4 a   /*****************************************************/
 5 a   fdtest:
 6 b       procedure options(main);
 7 b       declare
 8 b         fdiv2 entry(fixed(7),float) returns(float),
 9 b         i fixed(7),
10 b         f float;
11 b
12 c       do i = 0 by 1;
13 c          put skip list('100 / 2 **',i,'=',fdiv2(i,100));
14 c       end;
15 b
16 b   end fdtest;
```

**Listing 18-5. The FDTEST Program**

```
        title   'div by power of two (function)'
        public  fdiv2
        extrn   ?signal
;       entry:
;               p1 -> fixed(7) power of two
;               p2 -> floating-point number
;       exit:
;               p1 -> (unchanged)
;               p2 -> (unchanged)
;       stack:  p2 / (2 ** p1)
fdiv2:                  ;HL = .low(.p1)
        mov     e,m     ;low(.p1)
        inx     h       ;HL = .high(.p1)
        mov     d,m     ;DE = .p1
        inx     h       ;HL = .low(p2)
        ldax    d       ;a = p1 (power of two)
        mov     e,m     ;low(.p2)
        inx     h       ;HL = .high(.p2)
        mov     d,m     ;DE = .p2
        xchg            ;HL = .p2
;
;       A = power of 2, HL = .low byte of fp num
        mov     e,m     ;E = low mantissa
        inx     h       ;to middle of mantissa
        mov     d,m     ;D = middle mantissa
        inx     h       ;to high byte of mantissa
        mov     c,m     ;C = high mantissa
        inx     h       ;to exponent byte
        mov     b,m     ;B = exponent
        inr     b       ;B = 00?
        dcr     b       ;becomes 00 if so
        jz      fdret   ;to return from float div
dby2:   ;divide by two
        ora     a       ;counted power of 2 to zero?
        jz      fdret   ;return if so
        dcr     a       ;count power of two down
        dcr     b       ;count exponent down
        jnz     dby2    ;loop again if no underflow
;
;underflow occurred, signal underflow condition
        lxi     h,siglst;signal parameter list
        call    ?signal ;signal underflow
        lxi     b,0     ;clear to zero
        lxi     d,0     ;for default return
```

**Listing 18-6. FDIV2.ASM Assembly Language Program (8080)**

```
;
fdret:  pop     h       ;recall return address
        push    b       ;save high order fp num
        push    d       ;save low order fp num
        pchl            ;return to calling routine
;
        dseg
siglst: dw      sigcod  ;address of signal code
        dw      sigsub  ;address of subcode
        dw      sigfil  ;address of file code
        dw      sigaux  ;address of aux message
;       end of parameter vector, start of params
sigcod: db      3       ;03 = underflow
sigsub: db      128     ;arbitrary subcode for id
sigfil: dw      0000    ;no associated file name
sigaux: dw      undmsg  ;0000 if no aux message
undmsg: db      32,'Underflow in Divide by Two',0
        end
```

**Listing 18-6 (continued)**

```
;               Division by power of two (function)

        cseg
        public  fdiv2
        extrn   ?signal:near

;       entry:
;               p1 -> fixed(7) power of two
;               p2 -> floating point number
;       exit:
;               p1 -> (unchanged)
;               p2 -> (unchanged)
;       stack:  p2 /  (2 ** p1)

fdiv2:                  ;BX = .low(.p1)
        mov     si,[bx]         ;SI = .p1
        lods    al              ;AL = p1 (power of 2)
        mov     bx,2[bx]        ;BX = .p2

;       AL = power of 2, BX = .low byte of fp num

        mov     dx,[bx]         ;DX = low and middle mantissa
        mov     cx,2[bx]        ;CL = high mantissa, CH = exponent
        or      ch,ch           ;exponent zero?
        jz      fdret           ;to return from float div
```

**Listing 18-7. FDIV2.A86 Assembly Language Program (8086)**

```
dby2:                   ;divide by two
        or      al,al           ;counted power of 2 to zero?
        jz      fdret           ;return if so
        dec     al              ;count power of two down
        dec     ch              ;count exponent down
        jnz     dby2            ;loop again if no underflow

;       Underflow occurred, signal underflow condition

        mov     bx,offset siglst;signal parameter list
        call    ?signal         ;signal underflow
        sub     cx,cx           ;clear result to zero for default return
        mov     dx,cx

fdret:  pop     bx              ;recall return address
        push    cx              ;save high order fp num
        push    dx              ;save low order fp num
        jmp     bx              ;return to calling routine
        dseg
siglst  dw      offset sigcod   ;address of signal code
        dw      offset sigsub   ;address of subcode
        dw      offset sigfil   ;address of file code
        dw      offset sigaux   ;address of aux message
;       end of parameter vector, start of params
sigcod  db      3               ;03 = underflow
sigsub  db      128             ;arbitrary subcode for id
sigfil  dw      0000            ;no associated file name
sigaux  dw      offset undmsg   ;0000 if no aux message
undmsg  db      32,'Underflow in Divide by Two',0

        end
```

**Listing 18-7. (continued)**

## 18.3 Direct Operating System Function Calls

You can have direct access to all the operating system functions through the optional subroutines in assembly language programs which are included in source form on your PL/I sample program disk. The sample program disk also contains the file REL-NOTES.PRN which describes these assembly language programs and several PL/I programs that test the various function calls.

The subroutines in these programs are not included in the standard PLILIB.IRL file because specific applications might require changes to the system functions that either remove operations to decrease space, or alter the interface to a specific function. If the interface to a function changes, you must change the entry point to avoid confusion.

**Note:** be careful when you use these entry points instead of the normal PL/I facilities. For example, if you use the MEMPTR function to effect memory management, be aware that PL/I uses the dynamic storage area for processing RECURSIVE procedures and file I/O buffering. There is no guarantee that the dynamic storage area will not be used for other purposes as addtional facilities are added to PL/I.

Also, when you use the various file maintenance functions, such as DELETE(#19) or RENAME (#23), do not access a file that is currently open in the PL/I file system. Simple peripheral access, as shown in these examples, is generally safe because no buffering takes place.

*End of Section 18*

# Section 19
# Dynamic Storage and Stack Routines

This section describes some functions in the PL/I Run-time Subroutine Library (RSL) that perform dynamic memory management and manipulate the stack size.

## 19.1 Dynamic Storage Subroutines

The RSL includes a number of functions that provide access to the dynamic storage routines. These routines maintain a linked list of all unallocated storage. Upon request, these routines search for the first available segment in the free list that satisfies the request size, remove the requested segment, and return the remaining portion to the free list. If the storage is not available, the run-time system signals ERROR(7), Free Space Exhausted.

PL/I dynamically allocates storage upon entry to RECURSIVE procedures, when processing explicit or implicit OPEN statements for files performing disk I/O, or when processing an ALLOCATE statement. PL/I always allocates an even number of bytes or whole words, no matter what the request size.

### 19.1.1 The TOTWDS and MAXWDS Functions

It is often useful to find the amount of storage available at any given point while the program is running. The TOTWDS (Total Words) and MAXWDS (Max Words) functions provide this information.

You must declare the functions in the calling program as:

```
declare totwds entry returns(fixed(15));
declare maxwds entry returns(fixed(15));
```



When you invoke the TOTWDS subroutine, it scans the free storage list and returns the total number of words (double bytes) available. The MAXWDS subroutine returns the size (in words) of the largest contiguous segment in the free list. A subsequent ALLOCATE statement that specifies a segment size less than or equal to MAXWDS does not signal ERROR(7), because at least that much storage is available.

Both TOTWDS and MAXWDS count in word units, so the returned values can be held by FIXED BINARY(15) counters. Both TOTWDS and MAXWDS return the value -1 if they encounter invalid link words while scanning the free space list. This is usually due to an out-of-bounds subscript or pointer store operation. Otherwise, these functions return a nonnegative integer value.

### 19.1.2 The ALLWDS Subroutine

The PL/I Run-time Subroutine Library contains a subroutine, called ALLWDS, that you use to control the dynamic allocation size. You must declare the subroutine in the calling program as:

```
declare allwds entry(fixed(15)) returns(pointer);
```

The ALLWDS subroutine allocates a memory segment in words equal to the size given by the input parameter, and returns a pointer to the allocated segment. If no segment is available, ALLWDS signals the ERROR(7) condition. The input value must be a nonnegative integer.

Listing 19-1 shows the ALLTST program which is an example of how to use the TOTWDS, MAXWDS, and ALLWDS functions. Listing 19-2 shows a sample interaction with the ALLTST program.

```
 1 a   /*****************************************************/
 2 a   /* This program tests the TOTWDS, MAXWDS, and ALLWDS */
 3 a   /* functions from the Run-time Subroutine Library.   */
 4 a   /*****************************************************/
 5 a┌──alltst:
 6 b│      procedure options(main);
 7 b│      declare
 8 b│         totwds entry returns(fixed(15)),
 9 b│         maxwds entry returns(fixed(15)),
10 b│         allwds entry(fixed(15)) returns(pointer);
11 b│
12 b│      declare
13 b│         allreq fixed(15),
14 b│         memptr ptr,
15 b│         meminx fixed(15),
16 b│         memory (0:0) bit(16) based(memptr);
17 b│
18 c│    ┌──do while('1'b);
19 c│    │     put edit (totwds(),' Total Words Available',
20 c│    │               maxwds(),' Maximum Segment Size',
21 c│    │               'Allocation Size? ') (2(skip,f(6),a),skip,a);
22 c│    │     get list(allreq);
23 c│    │     memptr = allwds(allreq);
24 c│    │     put edit('Allocated',allreq,' Words at ',unspec (memptr))
25 c│    │             (skip,a,f(6),a,b4);
26 c│    │
27 c│    │     /* clear memory as example */
28 d│    │   ┌──do meminx = 0 to allreq-1;
29 d│    │   │     memory(meminx) = '0000'b4;
30 d│    │   └──end;
31 c│    └──end;
32 b│
33 b└──end alltst;
```

**Listing 19-1. The ALLTST Program**

```
A>alltst

 24470 Total Words Available
 24470 Maximum Segment Size
Allocation Size? 0

Allocated      0 Words at 28D6
 24468 Total Words Available
 24468 Maximum Segment Size
Allocation Size? 100

Allocated      100 Words at 28DA
 24366 Total Words Available
 24366 Maximum Segment Size
Allocation Size? 500

Allocated      500 Words at 29A6
 23864 Total Words Available
 23864 Maximum Segment Size
Allocation Size? 23865

ERROR (7), Free Space Exhausted
Traceback: 016D
A>
```

**Listing 19-2. Interaction with the ALLTST Program**

## 19.2 The STKSIZ Function

In PL/I, the program stack is placed above the code and data area, and below the dynamic storage area (TPA). The default size of the program stack is 512 bytes, but can be changed using the STACK(n) option in the main procedure heading.

The STKSIZ (Stack Size) function returns the current stack size in bytes. This function is particularly useful for checking possible stack overflow conditions, or in determining the maximum stack depth during program testing.

You must declare the STKSIZ function in the calling program as:

```
declare stksiz returns(fixed(15));
```

Listing 19-3 shows an example of the STKSIZ function in the program called ACKTST, where it checks the maximum stack depth during RECURSIVE procedure processing. Listing 19-4 shows an interaction with this program.

```
 1 a   /************************************************/
 2 a   /* This program tests the STKSIZ function while */
 3 a   /* evaluating a RECURSIVE procedure.            */
 4 a   /************************************************/
 5 a ┌─ack:
 6 b │     procedure options(main,stack(2000));
 7 b │     declare
 8 b │       (m,n) fixed,
 9 b │       (maxm,maxn) fixed,
10 b │       ncalls decimal(6),
11 b │       (curstack, stacksize) fixed,
12 b │       stksiz entry returns(fixed);
13 b │
14 b │     put skip list('Type max m,n: ');
15 b │     get list(maxm,maxn);
16 c │   ┌─do m = 0 to maxm;
17 d │   │   ┌─do n = 0 to maxn;
18 d │   │   │     ncalls    = 0;
19 d │   │   │     curstack  = 0;
20 d │   │   │     stacksize = 0;
21 d │   │   │     put edit('Ack(',m,',',n,')=',ackermann(m,n),
22 d │   │   │         ncalls,' Calls,',stacksize,' Stack Bytes')
23 d │   │   │         (skip,a,2(f(2),a),f(6),f(7),a,f(4),a);
24 d │   │   └─end;
25 c │   └─end;
26 b │     stop;
27 b │
28 b │   ┌─ackermann:
29 c │   │   procedure(m,n) returns(fixed) recursive;
30 c │   │
31 c │   │   declare
32 c │   │     (m,n) fixed;
33 c │   │   ncalls = ncalls + 1;
34 c │   │   curstack = stksiz();
35 c │   │   if curstack > stacksize then
36 c │   │      stacksize = curstack;
37 c │   │   if m = 0 then
38 c │   │      return(n+1);
39 c │   │   if n = 0 then
40 c │   │      return(ackermann(m-1,1));
41 c │   │   return(ackermann(m-1,ackermann(m,n-1)));
42 c │   └─end ackermann;
43 b │
44 b └─end ack;
```

**Listing 19-3. The ACKTST Program**

```
A>acktst

Type max m,n: 6,6

Ack( 0, 0)=      1        1 Calls,   4 Stack Bytes
Ack( 0, 1)=      2        1 Calls,   4 Stack Bytes
Ack( 0, 2)=      3        1 Calls,   4 Stack Bytes
Ack( 0, 3)=      4        1 Calls,   4 Stack Bytes
Ack( 0, 4)=      5        1 Calls,   4 Stack Bytes
Ack( 0, 5)=      6        1 Calls,   4 Stack Bytes
Ack( 0, 6)=      7        1 Calls,   4 Stack Bytes
Ack( 1, 0)=      2        2 Calls,   6 Stack Bytes
Ack( 1, 1)=      3        4 Calls,   8 Stack Bytes
Ack( 1, 2)=      4        6 Calls,  10 Stack Bytes
Ack( 1, 3)=      5        8 Calls,  12 Stack Bytes
Ack( 1, 4)=      6       10 Calls,  14 Stack Bytes
Ack( 1, 5)=      7       12 Calls,  16 Stack Bytes
Ack( 1, 6)=      8       14 Calls,  18 Stack Bytes
Ack( 2, 0)=      3        5 Calls,  10 Stack Bytes
Ack( 2, 1)=      5       14 Calls,  14 Stack Bytes
Ack( 2, 2)=      7       27 Calls,  18 Stack Bytes
Ack( 2, 3)=      9       44 Calls,  22 Stack Bytes
Ack( 2, 4)=     11       65 Calls,  26 Stack Bytes
Ack( 2, 5)=     13       90 Calls,  30 Stack Bytes
Ack( 2, 6)=     15      119 Calls,  34 Stack Bytes
Ack( 3, 0)=      5       15 Calls,  16 Stack Bytes
Ack( 3, 1)=     13      106 Calls,  32 Stack Bytes
Ack( 3, 2)=     29      541 Calls,  64 Stack Bytes
Ack( 3, 3)=     61     2432 Calls, 128 Stack Bytes
Ack( 3, 4)=    125    10307 Calls, 256 Stack Bytes
Ack( 3, 5)=
                .
                .
                .
                .
```

**Listing 19-4. Output From the ACKTST Program**

*End of Section 19*

# Section 20
# Overlays

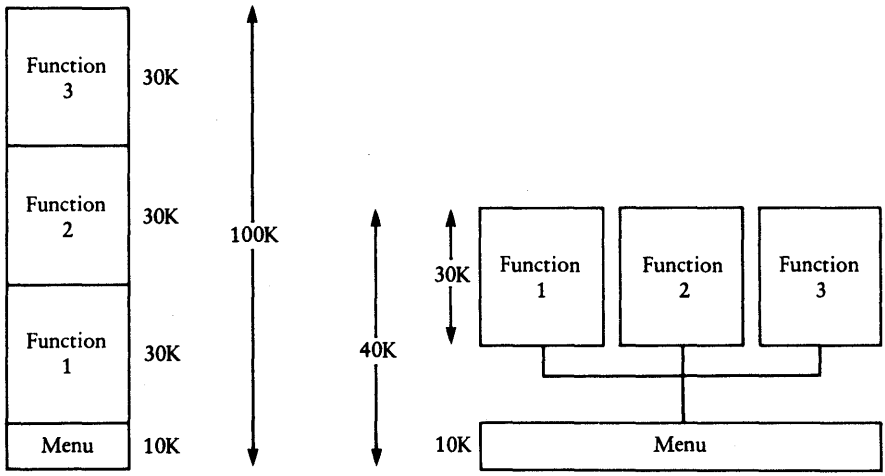
This section describes how to use the linkage editor to create PL/I overlays. Overlays are programs comprised of separate files. The advantage of overlays is that they share the same memory locations, so you can write large programs that run in a limited memory environment.

## 20.1 Using Overlays in PL/I

In both the 8080 and 8086 implementations, the size of the Transient Program Area (TPA) determines the upper limit on the size of a program. However, there is another constraint in the 8086 implementation. Although there can be enough memory space available on the system, the Compiler generates code that assumes the Small memory model. The Small model means that when you link one or more OBJ files with the Run-time Subroutine Library (RSL), the size of the code and data sections in the CMD are each limited to 64K. Thus, the Compiler determines the upper limit on the size of any program, but the size limit is not encountered until link time.

With modular design, you can write a large program that does not need to reside in memory all at once. For example, many application programs are menu-driven, in which the user selects one of a number of functions to perform. Because the functions are separate and invoked sequentially, they do not need to reside in memory simultaneously. When one of the functions is complete, control returns to the menu portion of the program, from which the user selects the next function. Using overlays, you can divide such a program into separate subprograms that can be stored on disk and loaded only when required.

The following figure illustrates the concept of overlays. Suppose a menu-driven application program consists of three separate user-selected functions. If each function requires 30K of memory, and the menu portion requires 10K, then the total memory required for the program is 100K, as shown in Figure 20-1a. However, if the three functions are designed as overlays, as shown in Figure 20-1b, the program requires only 40K, because all three functions share the same memory locations.




20-1a. Without Overlays 20-1b. Separate Overlays

**Figure 20-1. Using Overlays in a Large Program**

You can also create nested overlays in the form of a tree structure, where each overlay can call other overlays up to a maximum nesting level that the Overlay Manager determines. Section 20.3 describes the command line syntax for creating nested overlays.

Figure 20-2 illustrates the tree structure of overlays. The top of the highest overlay determines the total amount of memory required. In Figure 20-2, the highest overlay is SUB4. This is substantially less memory than would be required if all the functions and subfunctions had to reside in memory simultaneously.

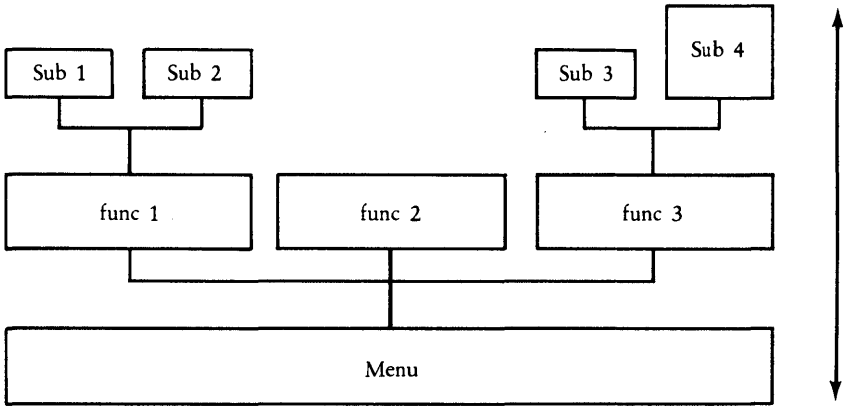



**Figure 20-2. Tree Structure of Overlays**

## 20.2 Writing Overlays in PL/I

There are two ways to write PL/I programs that use overlays. The first method involves no special coding, but has two restrictions. The first restriction is all that overlays must be on the default drive; the second is that the overlay names must be determined at translation time and cannot be changed at run-time.

The second method requires a more involved calling sequence, but does not have either of the restrictions of the first method.

### 20.2.1 Overlay Method One

To use the first method, you declare an overlay as an entry constant in the module where it is referenced. As an entry constant, the overlay can have parameters declared in a parameter list. The overlay itself is simply a PL/I procedure or group of procedures.

For example, the following program is a root module with one overlay:

```
root:
    procedure options(main);
    declare
       ovlay1 entry(character(15));
    put skip list('root');
    call ovlay1('overlay 1');
end root;
```

The overlay OVLAY1.PLI is defined as follows:

```
ovlay1:
      procedure(c);
      declare
         c character(15);
      put skip list(c);
end ovlay1;
```

**Note:** when passing parameters to an overlay, you must ensure that the number and type of the parameters are the same in both the calling program and the overlay.

When the program runs, ROOT first displays the message 'root' at the console. The CALL statement then transfers control to the Overlay Manager. The Overlay Manager loads the file OVLAY1 from the default drive and transfers control to it.

When the overlay receives control, it displays the message 'overlay 1' at the console. OVLAY1 then returns control directly to the statement following the CALL statement in ROOT. The program then continues from that point.

If the requested overlay is already in memory, the Overlay Manager does not reload it before transferring control.

The following constraints apply to overlay method one:

- The label in the call statement is the actual name of the overlay file loaded by the Overlay Manager; consequently, the two names must agree.
- The name of the entry point to an overlay need not agree with the name used in the calling sequence, but using the same name avoids confusion.
- The Overlay Manager only loads overlays from the drive that was the default when the root module began execution. The Overlay Manager disregards any changes in the default drive that occur after the root module begins execution.
- The names of the overlays are fixed. To change the names of the overlays, you must edit, recompile, and relink the program.
- No nonstandard PL/I statements are needed. Thus, you can postpone the decision on whether or not to create overlays until link time.

### 20.2.2 Overlay Method Two

In some applications, you might want to have greater flexibility with overlays, such as loading overlays from different drives, or determining the name of an overlay from the console or a disk file at run-time.

To do this, a PL/I program must declare an explicit entry point into the Overlay Manager, as follows:

```
declare ?ovlay entry(character(10),fixed(1));
```

This entry point requires two parameters. The first is a 10-character string that specifies the name of the overlay to load, and an optional drive code in the standard format (d:filename).

The second parameter is the Load Flag. If the Load Flag is 1, the Overlay Manager loads the specified overlay whether or not it is already in memory. If the Load Flag is 0, the overlay manager loads the overlay only if it is not already in memory.

Using this method, the example illustrating method one appears as follows:

```
root:
    procedure options(main);
    declare
       ?ovlay entry(character(10),fixed(1)),
       dummy entry(character(15)),
       name character(10);

    put skip list('root');
    name = 'OV1';
    call ?ovlay(name,0);
    call dummy('overlay 1');
end root;
```

The file OV1.PLI is the same as the previous example.

At run-time, the statement

```
call ?ovlay(name,0);
```

directs the Overlay Manager to load OV1 from the default drive (1 is the current value of the variable name); control then transfers to OV1. When OV1 finishes processing, control returns to the statement following the invocation.

In this example, the variable name is assigned the value 'OV1'. However, you could also supply the overlay name as a character string from some other source, such as the console keyboard.

The following constraints apply to overlay method two:

- You can specify a drive code so the Overlay Manager can load overlays from drives other than the default drive. If you do not specify a drive code, the Overlay Manager uses the default drive as described in method one.
- If you pass any parameters to the overlay, they must agree in number and type with the parameters that the overlay expects.

### 20.2.3 General Overlay Constraints

The following general constraints apply when creating overlays in a PL/I program:

- Each overlay has only one entry point. The Overlay Manager in the PL/I Run-time Subroutine Library assumes that this entry point is at the load address of the overlay.
- You cannot make an upward reference from a module to entry points in overlays higher on the tree. The only exception is a reference to the main entry point of the overlay. You can make downward references to entry points in overlays lower on the tree or in the root module.
- Common segments (EXTERNALS in PL/I) that are declared in one module cannot be initialized by a module higher in the tree. The linkage editor ignores any attempt to do so.
- You can nest overlays to a depth of five levels.
- The Overlay Manager uses the default buffer located at 80H, so user programs should not depend on data stored in this buffer. Note that in the 8086 implementation, the default buffer is at 80H relative to the base of the Data segment.

## 20.3 Command Line Syntax

To specify overlays in the command line of the linkage editor, enclose each overlay specification in parentheses. You can create overlays with LINK-80 in one of the following forms:

```
link root(ov1)

link root(ov1,part2,part3)

link root(ov1 = part1,part2,part3)
```

The first form produces the file OV1.OVL from the file OV1.REL. The second form produces the file OV1.OVL from OV1.REL, PART2.REL, and PART3.REL. The third form produces the file OV1.OVL from PART1.REL, PART2.REL, and PART3.REL.

Create overlays with LINK-86 using the same forms:

```
link86 root(ov1)

link86 root(ov1,part2,part3)

link86 root(ov1 = part1,part2,part3)
```

The first form produces the file OV1.OVR from the file OV1.OBJ. The second form produces the file OV1.OVR from OV1.OBJ, PART2.OBJ, and PART3.OBJ. The third form produces the file OV1.OVR from PART1.OBJ, PART2.OBJ, and PART3.OBJ.

In the command line, a left parenthesis indicates the start of a new overlay specification, and also indicates the end of the group preceding it. All files to be included at any point on the tree must appear together, without any intervening overlay specifications. You can use spaces to improve readability, but do not use commas to set off the overlay specifications from the root module or from each other.

For example, the following command line is invalid:

```
A>link root(ov1),moreroot
```

The correct command is as follows:

```
A>link root,moreroot(ov1)
```

To nest overlays, you must specify them in the command line with nested parentheses. For example, the following command line creates the overlay system shown in Figure 20-2:

```
A>link menu(func1 (sub1)(sub 2)) (func2) (func3 (sub3)(sub4
```

*End of Section 20*

# Index


## A

A format, 36, 115
actual parameter, 26, 104, 109, 233
aggregate data, 10
algorithms, 144, 197, 199, 222
ALLOCATE statement, 44, 129, 134
application programs, menu-driven, 267-8
arguments, 42, 64, 65
arithmetic data, 10
arrays, 13, 17, 231
ASCII character data, 31
assignment statement, 21, 92, 120, 126, 153, 168, 198
AUTOMATIC, 41

## B

B format, 36
B1 format, 13
B2 format, 13
B3 format, 13
B4 format, 13
BASED storage class, 42
BASED variable, 10, 43, 129-131
BASIC, 183
BCD, 186
BEGIN, 163
BEGIN block, 5, 49, 102, 132, 153
BIF, 2
Binary Coded Decimal (BCD), 186
binary exponent, 10
bit-string constant, 13
bit-string variables, 13
blank padding, 36
block, 5
block nesting, 5
block-structure, 29, 163
buffer, 77, 125, 126
buffer size, 84
built-in
  DECIMAL function, 205
  functions, 2, 12
  LOCK, 32
  MOD function, 198
  ROUND function, 198

## C

CALL statement, 6, 7, 26, 27, 153, 217, 218, 270
call statement, label in, 153, 217, 271
calls by reference, 26
case, 49, 120
CEIL function, 208
CHARACTER, 96, 207
CHARACTER variables, 13
character-string constants, 13
CLOSE statement, 84
COBOL, 183
code generation, 57
code optimization, 57
COLUMN, 36
command file, 54, 59
comments, 51
common segments, declared in one module, 273
Compiler
  options, 55
  overlays, 57

complement,
  2's, 186
  10's, 186, 187, 188
computational expressions, 22
computed GOTO, 101
condition
  categories, 38, 39, 60
  processing, 23, 37, 38, 111, 166
  Stack, 109
conditional branching, 23, 29
connected storage, 18
containing blocks, 5, 7, 29
context, 3
control
  characters, 2, 96, 216
  data, 13
  format items, 36
  variable, 24
cross-sectional reference, 17

## D

data
  aggregate, 16
  constants, 10
  conversion, 22, 96, 107, 127, 166, 183, 207
  format items, 36
  set, 29, 30
  structure, 80, 88, 92, 139
  variables, 10
debugging, 59
DECIMAL, 188
DECIMAL built-in function, 161, 208
DECIMAL function, 192, 193
declarations, 49, 50
declarative statements, 3, 9
DECLARE keyword, 50
DECLARE statement, 10
default buffer, 273
default drive, changing, 272
default values, 12
delete, 32
DEMO program, 59
dimension array, 43
DIRECT attribute, 93
DIRECT files, 31
DIVIDE built-in functions, 188, 196
DO-group, 7, 49, 71, 83, 93, 96, 101, 105, 108-109, 120, 123, 125, 126, 134-135, 142, 153-154, 166
double-precision number, 10-11
downward reference to entry point, 273
drive code, 272
dynamic memory management, 129, 261

## E

E format, 36
EDIT formats, 197
EDIT-directed I/O, 35
ENDPAGE condition, 115
entry constant, 7, 14, 177
ENTRY constants, 14, 174-176, 270
ENTRY data, 13, 14
entry point, explicit, 271
ENTRY variables, 14, 174-176, 231
environment, 3, 5, 7, 27, 38, 113, 132
ENVIRONMENT
  attribute, 32
  option, 88
ERROR condition, 216
error messages 49, 57, 58
executable statements, 3, 10, 57
explicit declaration, 10

expression, 21
EXTERNAL attribute, 174, 175
external
  device, 16, 29, 60
  procedures, 5, 7, 15

## F

F format, 36
file
  access methods, 35
  constant, 16, 30
  data, 16
  Descriptor, 33
  Parameter Block (FPB), 33
  variable, 16, 27
FILE variables, 175
file-handling statements, 29, 32
file_id, 30, 32, 35
FIXED BINARY, 10, 161, 173, 215
  data, 10, 158
FIXED DECIMAL, 11, 74, 96, 185, 186, 197, 203, 204, 215, 217
  data, 11
FIXEDOVERFLOW, 158, 192
FIXED OVERFLOW, 188, 216
fixed record size, 32, 88
FLOAT BINARY, 10, 158, 185, 203, 205, 207
  data, 10, 74, 184
formal parameter, 26, 175-176, 233
format items list, 35
FORMAT statement, 37
FORTRAN, 183
FREE statement, 44, 129, 136
free storage area, 44
free-format language, 49
function procedure, 7, 26
function reference, 6, 7, 27, 153

## G

GET EDIT, 35
  statement, 115, 125, 127
GET LIST, 35, 82
  statement, 35, 80, 93, 102, 142
GOTO, 101, 103
  statement, 29, 104, 109, 110

## H

halting the Compiler, 58
hierarchical structure, 3
high-level language, 1

## I

IF statement, 29, 46, 50
implicit declaration, 10, 13, 14
implied attributes, 34
implied base, 129, 131
%INCLUDE statement, 45, 80
indentation, 49
INDEX function, 120, 232
INITIAL attribute, 41, 42
INPUT file, 31
integers, 10
internal
  buffer, 35, 84
  buffer sizes, 32, 84
  procedure, 7, 15
  representation, 43, 183
  stack, 61
invoking Compiler, 55
iteration, 23

## K

key, 32, 40, 91
KEYED
  attribute, 31, 88, 91
  file, 32, 91, 92
KEYTO option, 91
keywords, 50

## L

label constants, 14, 102, 105
LABEL data, 14
label variables, 14, 102, 105
level, 32
LINE, 36
line-directed, 35
linemark, 31
LINESIZE attribute, 32
LINK-80, 273
LINK-86, 274
linkage editor, creating overlays with, 273
list processing, 132
list-directed, 35
load address, 273
load flag, 271
local reference, 104
locked mode, 32
logical units, 3-4, 14, 27

## M

MAIN option, 161
main
  procedure, 5
  structure, 19
mantissa, 10
mathematical functions, 12
member, 19
menu-driven application programs, 267-8
modular design, 267
module, upward reference from, 273
multiple data items, 10

## N

native code, 57
nested overlays, 268
nesting levels, 58
  maximum, 268
nesting overlays, 273, 274
noncomputational expressions, 22
nonlocal reference, 104
null
  pointer, 141, 143
  statement, 29, 46

## O

object
  code, 56
  file, 54, 59
ON ENDFILE statements, 114
ON ENDPAGE, 115
ON
  condition, 38, 109-110, 112, 216
  statement, 38, 109, 110, 114
ON-body, 38, 110
ON-unit, 112-117, 120, 142, 166
ONCODE function, 41
ONFILE function, 41
ONKEY function, 41
open mode, 32
OPEN statement, 30, 31, 35, 80, 84, 88, 93, 216, 233

OUTPUT file, 31
overlay
  entry point, 271, 273
  manager, 270-3
  method one constraints, 271
  method two constraints, 272
  name of entry point to, 271
  names, when determined, 269
  SUB4, 269
  specifications, 274
overlays
  changing names of, 271
  composition of, 268
  creating with LINK-80, 273
  creating with LINK-86, 274
  enclosing in parentheses, 273
  flexibility with, 271
  general constraints, 273
  higher on tree, 273
  left parenthesis in, 274
  lower on tree, 273
  method one, 270
  method two, 271
  nested, 268
  nesting, 273, 274
  passing parameters to, 270
  referencing entry points of, 273
  restrictions to, 269
  storing on disk, 267
  tree structure of, 268-9
  use of, 267
  using in a large program, 268
  when to create, 271
  writing, 269

## P

PAGE, 37
pagemark, 31
PAGESIZE attribute, 32
parameter list, 270
parameter passing, 26-27, 270, 272
parameters, agreeing with overlay, 272
parse function, 123
PASCAL, 183
pass
  1, 57
  2, 57
  3, 57
  by reference, 26
  by value, 26, 153
password protection, 32
Picture edit format, 208, 217
pointer
  data, 16, 42
  qualifier, 42, 130
  variable, 16, 42, 131-132
pointer-qualified reference, 42
precision, 11-12, 155, 186-196, 206-208
predefined file constants, 37
preprocessor statements, 45
PRINT, 31, 37, 41, 80, 85, 112, 216
PROCEDURE block, 5, 49, 104, 163
procedure
  body, 6
  definition, 26
  header, 6
  heading, 177
  invocation, 7, 23, 26
  name 6, 175
program
  development, 53
  maintenance, 6, 51, 101
program size, upper limit, 267
PUT EDIT statements, 35, 115
PUT LIST statements, 35, 85, 114, 216

## R

R, 37
read, 32
READ statement, 35, 91
READ with KEY statement, 93
Read-Only, 32
RECORD
  file, 32
  I/O, 35
recursive, 26, 36, 153, 154, 161
relative record, 32
  number, 91
remote format, 37
%REPLACE statement, 46, 71, 216, 222
RETURN statement, 155
RETURNS attribute, 176
REVERT statement, 38, 109-110
root module, 271
  with one overlay, 270
run-time stack, 161
Run-Time Subroutine Library (RSL) 53, 59, 129, 173, 205, 267

## S

scalar data items, 10
scalar value, 7
scale, 11, 190-196, 207-208
saving memory with overlays, 267
sequence control statements, 23
SEQUENTIAL files, 31
shared, 32
SIGNAL statement, 38-40, 108, 110, 115
single-precision number, 10
size of programs, upper limit, 267
SKIP, 37, 115
small memory model, 267
source file, 51, 53
special characters, 2
special forms, 57
stack, 187, 188, 193-194
STACK option, 161
standard, 42
STATIC attribute, 41, 174
STOP statement, 71, 166
storage class, 41
storage sharing, 26
STREAM file, 31, 80, 91-92, 112, 145
STREAM I/O, 35
string
  processing, 119, 123
  variables, 12
  variables, 12
structural statements, 1, 3
structure, 18, 132-136, 139-140, 142, 174
structured language, 1
subcode, 38, 39, 60, 107
subroutine, 50-51, 123, 133, 173, 205
subroutine procedures, 7, 26
subscripts, 14, 17
Subset G, 42, 192
SUBSTR, 119, 120, 127
SYM file, 53
Symbol Table, 53, 55, 56, 57, 59
SYSIN, 37, 61, 110, 111, 114
SYSPRINT, 37, 71, 85
system files, 54

## T

temporary variables, 57
TITLE attribute, 32
token, 125-127, 165-166
traceback, 60-61

Transient Program Area, 57, 129, 267
TRANSLATE function, 119
tree structures, 57
TRUNC, 49
truncation, 36, 74, 184-185, 206
  error, 11

## U

unconditional branching, 23, 29
UNLOCK functions, 32
UPDATE file, 31
upward reference to entry point, 273

## V

VARYING attribute, 13
vector, 103, 105
VERIFY function, 119, 125-127

## W

wildcard reference, 32
Write, 32
WRITE statement, 35, 88, 96
WRITE with KEYFROM statement, 93

# Reader Comment Form

We welcome your comments and suggestions. They help us provide you with better product documentation.

Date _____ Manual Title _____ Edition _____

1. What sections of this manual are especially helpful?

_____

_____

_____

_____

2. What suggestions do you have for improving this manual? What information is missing or incomplete? Where are examples needed?

_____

_____

_____

_____

3. Did you find errors in this manual? (Specify section and page number.)

_____

_____

_____

_____

COMMENTS AND SUGGESTIONS BECOME THE PROPERTY OF DIGITAL RESEARCH.

NO POSTAGE
NECESSARY
IF MAILED IN THE
UNITED STATES

**BUSINESS REPLY MAIL**

FIRST CLASS / PERMIT NO. 182 / PACIFIC GROVE, CA

POSTAGE WILL BE PAID BY ADDRESSEE

DIGITAL RESEARCH™

P.O. Box 579
Pacific Grove, California
93950

**Attn: Publication Production**