# PL/I

# Reference Manual

093-000204-00

*For the latest enhancements, cautions, documentation changes, and other information on this product, please see the Release Notice (085-series) supplied with the software.*

Ordering No. 093-000204
©Data General Corporation, 1978
All Rights Reserved
Printed in the United States of America
Revision 00, March 1978
Licensed Material - Property of Data General Corporation

PL/I Reference Manual
093-000204

Revision History:

Original Release - March 1978

# Preface

The PL/I Reference Manual is designed for use by experienced PL/I programmers. If you have never programmed in PL/I, we suggest that you read our tutorial manual, entitled Plain PL/I (093-000216).

Data General PL/I is a subset of American National Standard Programming Language PL/I (ANSI X3.53-1976) with the following extensions:

- %REPLACE statement
- DO CASE statement
- ASCII, RANK, and SIZE built-in functions
- READ and WRITE statements with STREAM files
- Interactive I/O with your console.

Data General PL/I allows you to call assembly language procedures from your programs. You may also use the SYS library function to call any operating system function.

We use the following conventions in examples in this manual:

| **Where** | **Means** |
|---|---|
| KEYWORD | You must use the keyword (or its accepted abbreviation) as shown. |
| required | You must give some argument (such as a filename). |
| *[optional]* | You have the option of entering some argument. Don't enter the brackets; they only set off what's optional. |
| ... | You may repeat the preceding entry or entries. The explanation will tell you exactly what you may repeat. |

Additionally, we use certain symbols in special ways:

| **Symbol** | **Means** |
|---|---|
| ) | Press the NEW-LINE or RETURN key on your terminal's keyboard. |
| □ | Represents a space. (We use this only when we must; normally, you can see where spaces appear.) |

All numbers are decimal.

Finally, we usually show all examples of entries and system responses in THIS TYPEFACE. But, where we *must* clearly differentiate your entries from system responses in a dialog, we will use

THIS TYPEFACE TO SHOW YOUR ENTRY)
*THIS TYPEFACE FOR THE SYSTEM RESPONSE*

End of Preface

# Contents

# Chapter 1 The Structure of a PL/I Program

## 1.1 The Source Text of a PL/I Program

### Identifiers

An *identifier* is either a keyword or a declared name. The rules for writing identifiers are:

a. An identifier must contain 1 to 32 characters chosen from the upper- and lowercase letters of the alphabet, digits, and the break character _ (underscore). An identifer may not contain blanks.

b. The first character must be a letter.

Examples:

```
X.
YEAR2 .
RATE_OF_PAY
Last_Name
```

A *keyword* is an identifier that has a specific meaning in the PL/I language when you use it in its intended context. A *declared name* is an identifier that you define in your own program.

Example:

```
IF YOUR_INTEGER=2 THEN GOTO YOUR_SPOT;
```

The example assumes that you define YOUR_INTEGER and YOUR_SPOT somewhere in the program, but that IF, THEN and GOTO are predefined in the PL/I language; they have a certain meaning when written in this statement in this form. A complete list of keywords is given in Appendix A.

You may use any keyword as a declared name. However, such usage may make the program difficult to read.

A keyword may consist of uppercase or lowercase letters, but the two are equivalent. For example: DECLARE and declare are the same keyword.

For user-declared names, however, the upper- and lowercase letters are not equivalent. Thus, LAST_NAME is regarded as different from Last_Name.

### Literal Constants

A literal constant may be:

- an arithmetic constant, such as

  0504, 3.14159, or 1.23E+6

- a character-string, such as

  "May 27, 1938"

- a bit-string, such as

  "01100010"B

A literal constant has only the specific value that you give it when you write it in the program; it cannot change during execution of the program. Sections 2.1 and 2.2 define and give more examples of literal constants.

## Special Symbols, Delimiters

A number of special symbols are used in PL/I. Examples are the arithmetic operators such as + and -, the relational operators such as = and < =, and others such as ; and :. Each of these characters or character pairs has its own meaning defined in the PL/I language much like keywords. The use of these special symbols is explained wherever appropriate throughout the manual. A complete list of them is given in Appendix B.

Identifiers and arithmetic constants are separated from one another by one or more delimiters.

A *delimiter* may be a graphic delimiter, a space, a comment, a bit-string constant, a character-string constant, or a PICTURE. (PICTURE is explained in Section 2.1.)

The *graphic delimiters* are:

| | | | | | | | |
|---|---|---|---|---|---|---|---|
| + | - | * | / | ** | ^ | & | ! |
| !! | : | ; | ( | ) | , | . | -> |
| = | ^= | <= | >= | > | < | ^> | ^< |

A *space* may be an ASCII blank, new line, form feed, or horizontal tab character.

*Comments* are used to provide explanatory information in the source program. You may use a comment anywhere a delimiter is appropriate. A comment is initiated by the character pair /* and terminated by the character pair */. It may be of any length, even spanning multiple lines. You may include any characters in a comment except the comment starter sequence /* or terminator sequence */.

Spaces and comments may be used freely between any two identifiers, literal constants, or other delimiters.

Examples:

```
A+B+C
DO I = a to 1000;
DECLARE A BIT (19), B POINTER;
/*THIS IS A COMMENT*/
```

## Source Text Insertion and Replacement

The %INCLUDE and %REPLACE statements are source manipulation commands recognized by the PL/I compiler. The text that results from the expansion of all %INCLUDE and %REPLACE statements must be a valid source program.

### %Include

You use the %INCLUDE statement to include source text from a file other than the source file in your program during compilation.

The %INCLUDE statement looks like this:

```
%INCLUDE "filename";
```

This statement causes the complete contents of the file named in the quoted string to be counted as part of the source when your program is compiled.

For example, if you have a complicated declaration which you want to use in several procedures, you may put the declaration in a separate file and use %INCLUDE to insert the declaration wherever you needed it. If the declaration is:

```
DECLARE 1 PASS STATIC EXTERNAL,
                         2 NAME CHARACTER (20),
                         2 FLIGHT CHARACTER (20),
                         2 SEAT CHARACTER (2);
```

and you have it in an AOS file called PASSENGER, you insert it into a program like this:

```
DECLARE P POINTER;
%INCLUDE "PASSENGER";
DECLARE X FIXED;
```

When the program is compiled, the text compiled would be:

```
DECLARE P POINTER;
DECLARE 1 PASS STATIC EXTERNAL,
                         2 NAME CHARACTER (20),
                         2 FLIGHT CHARACTER (20),
                         2 SEAT CHARACTER (2);
DECLARE X FIXED;
```

### %Replace

The form of the %REPLACE statement is:

```
%REPLACE identifier BY constant;
```

The constant may be a character-string constant, a bit-string constant, or an arithmetic constant optionally preceded by a minus sign.

You use the %REPLACE statement to replace all further instances of the given identifier by the specified constant. This can help make your program easy to modify.

For example, if you are not sure what size you want an array to be, you can use an identifier for the size of the array and then use %REPLACE to give the array the size you want for that version of the program:

```
UNCERTAIN:
        PROCEDURE;

        %REPLACE TABLESIZE BY 10;

        DECLARE ARRAY (TABLESIZE) FIXED BINARY;
        DECLARE I FIXED BINARY;

        DO I = 1 TO TABLESIZE;
        ARRAY(I) = 1;
        END;
```

When you compile this program, the compiler will replace all succeeding instances of TABLESIZE with 10. If you want the array to have a different size, all you have to do is change the value in the %REPLACE statement.

A given identifier may appear in only one %REPLACE statement in a single source program.

## 1.2 Program Organization

### Procedures and Begin Blocks

A fundamental concept in the organization and structure of PL/I programs is the concept of blocks. There are two kinds of blocks: procedures, and begin blocks. In the source program, a *block* is simply a sequence of statements clearly delimited by a head statement and an END statement. The purpose of a block is to mark off a portion of the program in which names can be declared without interfering with other parts of the program. In more technical terms, you use blocks to delimit the scope of user-declared names. This is explained fully in Section 4.1.

Procedures have an even more important role in organizing the flow of control of the program. Begin blocks do not have this additional use.

A PL/I source program is called an external procedure. An *external procedure* has the following form:

```
label:    PROCEDURE [options];
          .
          .
          .
          END;
```

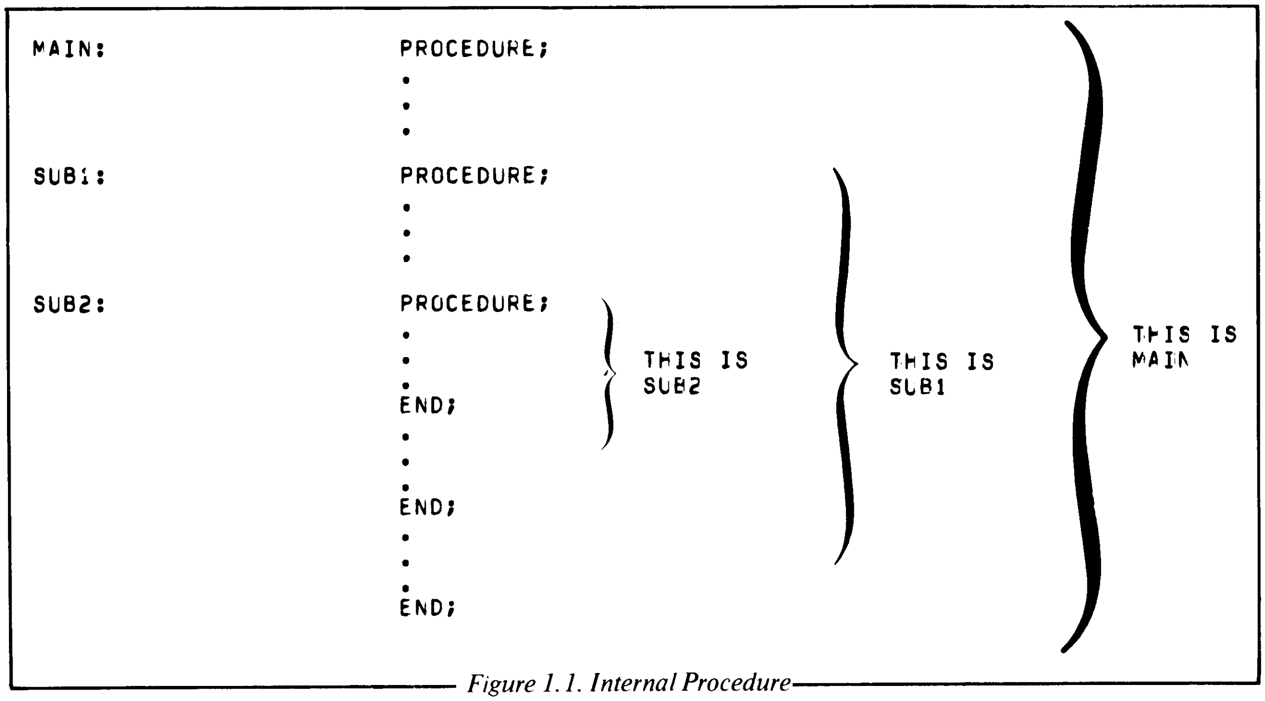
The label is the name used to reference the procedure; the details of the PROCEDURE statement are explained in Section 9.19.

A PL/I object program consists of one or more separately compiled external procedures. After all external procedures have been separately compiled, they are loaded together as one program. One of the external procedures is the main program; the others are subroutines or functions.

An external procedure can contain other procedures nested within itself. These procedures are called *internal procedures.*

You may also nest internal procedures within other internal procedures. Each procedure is bracketed by its own PROCEDURE and END statements, just as you would nest expressions within parentheses. Internal procedures are used as subroutines or functions.

Example:



```
MAIN:          PROCEDURE;
               .
               .
               .
SUB1:          PROCEDURE;
               .
               .
               .
SUB2:          PROCEDURE;
               .
               .
               .
               END;
               .
               .
               .
               END;
               .
               .
               .
               END;
```

*Figure 1.1. Internal Procedure*

A *begin block* consists of a BEGIN statement, a sequence of statements, and an END statement.

For example:

```
BEGIN;
.
.
.
END;
```

Like procedures, begin blocks can be nested. They can also be nested within procedures and can contain procedures.

The following discussion of nesting is important in order to understand the rules governing the scope of a name given in Section 4.1.

All of the text of a begin block except the label of the BEGIN statement is *contained* in the begin block.

Example:

```
A:          BEGIN;
            .
            .
            .
            END;
```

The underlined text is contained in begin block A.

All of the text of a procedure, except the label of the PROCEDURE statement and the labels of each of its ENTRY statements (if any), is contained in the procedure.

Example:

```
A:          PROCEDURE;
            .
            .
            .
B:          ENTRY;
            .
            .
            .
            END;
```

The underlined text is contained in procedure A.

The text contained in block A, but not contained in any inner block of A, is said to be immediately contained in block A.

For example, in procedure P below, B is contained in X and P, and B is immediately contained in X.

```
P:          PROCEDURE;
            .
            .
            .
X:          PROCEDURE;
            .
            .
            .
B:          BEGIN;
            .
            .
            .
            END; /* OF B*/
            END; /* OF X*/
            END; /* OF P*/
```

## Statements and Groups

A *statement* consists of a sequence of identifiers, constants, operators, and punctuation ending with a semicolon. A *simple statement* is one that contains no other statements. A *compound statement* contains other statements.

Examples:

```
DECLARE A FIXED DECIMAL(3);
DO I = 1 TO 5 BY 2;
A = B + I(6);
IF X > Y THEN GOTO P;
     ELSE Y = X;
```

The types of statements available are shown below.

Statements that control storage allocation:

ALLOCATE
DECLARE
FREE

Statement that assigns values to variables:

assignment statement

Input/Output statements:

CLOSE
DELETE
FORMAT
GET
OPEN
PUT
READ
REWRITE
WRITE

Statements used to define program structure:

BEGIN
DO
DO CASE
END
ENTRY
PROCEDURE
null statement

Flow control statements:

CALL
GOTO
IF
ON
RETURN
REVERT
SIGNAL
STOP

The IF, DO CASE and ON statements are compound statements.

All of the statements are defined in Chapter 9.

A *group* is a sequence of statements used to define the flow of control during program execution. The form of a group is:

```
DO [options];
.
.
.
END;
```

You may use a group anywhere you can use a single statement. For example, the IF statement may contain two other statements and either of these two statements may be a group.

```
IF A = B
      THEN GOTO P;
      ELSE DO;
                X=1;
                Y=2;         /* This group serves as
                END;         the ELSE clause*/
```

A group may be an iterative group or a noniterative group. An *iterative group* is a group beginning with an iterative DO statement (DO increment, DO list, or DO WHILE) specifying repetitive execution of the statements within the group. A *noniterative* group is a group beginning with a noniterative DO statement (simple DO or DO case).

Example:

```
DO I = 1 TO 100;
.
.
.
END;
```

### Labels

A *label* is a name that identifies a statement. You write the label as an identifer preceding the statement; the label must have a colon after it. Any statement other than a declare statement, a statement forming the THEN or ELSE clause of an IF statement, or a BEGIN statement that is the start of an ON unit, can have a label.

Example:

```
LBL: Y = Z;
```

A label may also be an identifier subscripted by a single optionally signed integer constant:

```
X(-2):
X(0):
X(1):.
```

Every ENTRY, PROCEDURE, and FORMAT statement must have a single unsubscripted label. Other statements, except DECLARE statements, can have up to 7 optionally subscripted labels.

For example:

```
A:    B:    C:    CALL F(X);
```

The appearance of an unsubscripted label in the text of the program declares it as a label constant, format constant, or entry constant according to the type of statement on which it appears as explained in Section 4.2.

All of the subscripted labels having the same name and appearing in the same block constitute a declaration of a label array constant as explained in Section 4.2.

## 1.3 The Flow of Control

As your PL/I program is executed, the processor executes the statements in a sequence called the *flow of control.* You define the flow of control by the sequential ordering of the statements in your program and by the use of various control transfer statements (such as the CALL statement). A program cannot create multiple paths for control to follow simultaneously, nor can it determine the real time rate at which it is executed.

When you bind one or more external procedures together as your object program, you indicate which procedure is to be the main procedure (i.e., the one in which execution starts).

### Block Activation

A block is *activated* when control enters the block and it remains active until control returns from the block. At least one block is always active. Since blocks may be nested and procedures may call each other, it is possible for several blocks to be active at the same time.

The order in which the blocks are activated determines the relationship between the blocks. If control passes from an active block A to block B, then A is said to be the predecessor of B, and B is the descendant of A.

Information about the allocation of data storage for a block activation is given in Chapter 3.

The flow of control in an activated block proceeds from statement to statement in the order in which the statements appear in the text of the block, unless a flow control statement transfers control elsewhere.

## The Flow of Control Between Blocks

Control enters a procedure only when one of the procedure's entry points is invoked by a function reference or a CALL statement. Control returns from the procedure by the execution of a RETURN statement, the execution of the END statement which terminates the procedure, or by the execution of a nonlocal GOTO statement. The rules governing the execution of a procedure are given in Sections 5.5 (Function Reference), 9.4 (CALL), 9.9 (END), 9.14 (GOTO), and 9.22 (RETURN).

A procedure is not executed when the normal sequential flow of the program reaches its PROCEDURE statement. In this case the procedure is simply skipped. For example, in the program below, the statement executed after C is G.

```
A:          PROCEDURE;
            .
            .
C:          X = 3
D:          PROCEDURE;
            .
            .
F:          END; /* OF D*/
G:          X = X + 1;
            .
            .
I:          END; /* OF A*/
```

Control enters a begin block by passing from the preceding statement through the BEGIN statement which heads the block. Control leaves a begin block by passing through the END statement that terminates the block, or by the execution of a statement that explicitly transfers control out of the block (a RETURN statement or nonlocal GOTO). You do not invoke a block by function references or CALL statements.

Example:

```
A:          PROCEDURE;
B:          DECLARE X FIXED BINARY;
C:          X = 3
D:          BEGIN;
E:          X = X*2;
F:          END; /* OF D*/
G:          X = X + 1;
H:          END; /* OF A*/
```

In this program the statements would be executed in the order in which they are written. The final value of X is 7.

## Recursive Activation

An active procedure may call itself or may be called by another active procedure. This reactivation is called *recursion.* You may invoke a procedure recursively only if you specify the RECURSIVE option in its PROCEDURE statement.

For example:

```
ABC:    PROCEDURE RECURSIVE;
        .
        .
        X = X+1;
        .
        IF X < 5 THEN CALL ABC;
FIN:    .
        .
        .
        END;
```

In this example statement, FIN is not reached until X>=5.

```
MAIN:   PROCEDURE
DECLARE OUT FILE;                               /* OUTPUT FILE */
DECLARE K FIXED BINARY;         /* SET UP AN INTEGER VARIABLE */

DO K = 1 TO 6;                          /* DO THIS LOOP 6 TIMES */
PUT FILE (OUT) SKIP LIST ("FACTORIAL OF "!!K!!" IS "!!FACT(K));
END;
STOP;                           /* THEN TERMINATE */
/* FUNCTION USED IN ABOVE LOOP FOLLOWS */

FACT:   PROCEDURE (N) RETURNS (FIXED BINARY) RECURSIVE;

DECLARE N FIXED BINARY;

IF N <= 1
        THEN RETURN (1);
        ELSE RETURN (N*FACT(N-1));
END;            /* FACT */

END;            /* MAIN */
```

Figure 1-2. A Recursive Procedure

In this example, MAIN is an external procedure containing a recursive internal procedure FACT. MAIN writes these six lines on the console:

```
FACTORIAL OF 1 IS 1
FACTORIAL OF 2 IS 2
FACTORIAL OF 3 IS 6
FACTORIAL OF 4 IS 24
FACTORIAL OF 5 IS 120
FACTORIAL OF 6 IS 720
```

When K = 4, the flow is as follows:

(a) the PUT statement calls FACT (4);
(b) the FACT procedure calls itself to calculate 4*FACT (3);
(c) FACT calls itself again to calculate 3*FACT (2);
(d) FACT calls itself to calculate 2*FACT (1);
(e) the call at (d) returns FACT(1) = 1;
(f) this enables the call at (c) to return FACT(2) = 2;
(g) this enables the call at (b) to return FACT(3) = 6;
(h) at last the call at (a) returns FACT(4) = 24.

## 1.4 Errors and ON Units

When a runtime error occurs during execution of a PL/I program, for example if you attempt to open a file for INPUT and the file does not exist, the PL/I default error handler, or *ON unit,* will write a message to the user console, and the program will abort.

You can override the default ON unit by supplying a user-defined ON unit within your program. This will cause transfer to a user-defined block of code whenever specified error conditions occur. For discussion on how to use ON units, see the sections on the ON, REVERT, and SIGNAL statements in Chapter 9 and the ONCODE built-in function in Chapter 10.

End of Chapter

# Chapter 2
# Data and Data Types

Data is represented in the text of a PL/I program by constants and variables. A *constant* is a data item that contains one value which never changes during the course of the program. A *variable* is a data item that can represent different values during program execution.

Every data item has an associated data type. The data type determines how the value represented by the data item is stored within the computer, and determines which operations can be performed on the value.

An *elementary data item* is a constant or variable that represents one value (at one time). Data can also be organized into arrays, structures, and arrays of structures (Section 2.7). These groupings, called data *aggregates,* represent more than one value. A single data item, whether an elementary data item or an element of an array or structure, is called a *scalar.*

Data is stored in multiples of 16-bit words, 8-bit bytes (2 per word), or single bits, depending on the *alignment* of the data type of the value. Word-aligned data must always begin on a word boundary, byte-aligned data may begin on a word or byte boundary, and bit-aligned data may begin on any bit. All elementary data items begin on a word boundary regardless of their data type or alignment.

The number of words, bytes, or bits needed to represent a value or data aggregate is called its *storage requirement.*

Data General PL/I supports the following data types:

| | |
|---|---|
| FIXED BINARY<br>FIXED DECIMAL<br>FLOAT BINARY<br>PICTURE | Arithmetic Data |
| CHARACTER<br>ALIGNED CHARACTER<br>VARYING CHARACTER<br>BIT<br>ALIGNED BIT | String Data |
| POINTER<br>LABEL<br>ENTRY<br>FILE | |

## 2.1 Arithmetic Data

The arithmetic data types are: FIXED BINARY, FLOAT BINARY, FIXED DECIMAL, and PICTURE. All numeric values have one of these data types. Every numeric value has an associated *precision* (p) which specifies how many binary or decimal digits the number contains. FIXED DECIMAL and PICTURE values also have an associated *scale factor* (q), which indicates the number of digits to the right of the decimal point.

The precision and scale of an arithmetic variable define the range of values that the variable can contain. The precision and scale of a variable can be explicitly specified when the variable is declared (see Chapter 4 on Declarations) or, if not given in the declaration, there is a default precision and scale which will be given to the variable.

The text of a PL/I program can contain unsigned FIXED BINARY, FLOAT BINARY, and FIXED DECIMAL constants. (A signed number such as -55 is considered to be an expression, not a constant.)

## Fixed Binary(p)

A FIXED BINARY(p) value is an integer that has a sign and p binary digits. A FIXED BINARY(p) value, r, lies in the range -(2**p)-1 < =r < =(2**p)-1. For example, a FIXED BINARY(15) value lies in the range -32767 < =r < =32767.

A FIXED BINARY variable can be declared with any precision from 1 to 15. If the precision is less than 15, the variable may still contain values of up to 15 bits, and the entire value will be used in any computation involving the variable. However, only as many bits as the precision specifies will be output if the variable is used in an output statement.

If your program assigns a non-integer arithmetic value to a FIXED BINARY variable, the value will be truncated. If your program assigns an arithmetic value greater than 32767 or less than -32767 to a FIXED BINARY variable, the variable will contain an unmeaningful value. (However, an expression involving only FIXED BINARY values may result in the number -32768.)

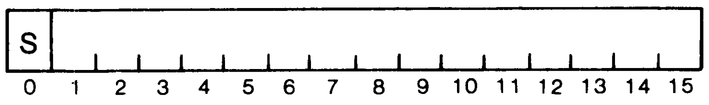
A FIXED BINARY constant is written in the text of a program as a decimal integer without a decimal point, such as: 321, 10, and 8214. However, if used in a context which does not specifically expect a FIXED BINARY value, a decimal integer will be interpreted as a FIXED DECIMAL constant.

Examples of contexts in which integer constants are taken as fixed binary are:

- an integer used as a subscript.
- an integer used in an arithmetic operation with a FIXED BINARY variable.

| | |
|---|---|
| Precision: | 1 < =p < =15 |
| Default Precision: | 15 |
| Alignment: | word |
| Storage Requirement: | 1 word |
| Internal Representation: | 15-digit 2's complement binary number. (See the *Programmer's Reference Manual ECLIPSE-line Computer.)* |

**Fixed Binary**



## Fixed Decimal(p,q)

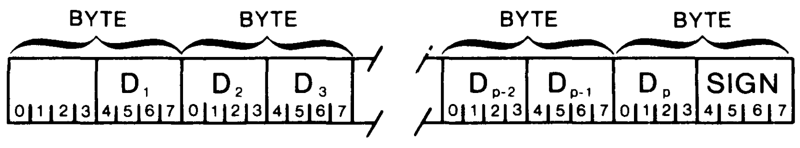
A FIXED DECIMAL(p,q) value has a sign and p decimal digits, q of which are to the right of the decimal point. A FIXED DECIMAL(p,q) value, r, lies in the range -10**(p-q) < r < 10**(p-q). For example, a FIXED DECIMAL (3,1) value lies in the range -99.9 < =r < =99.9.

Values that have a scale greater than the scale of a FIXED DECIMAL variable will be truncated if they are assigned to the variable. It is an error to assign to a FIXED DECIMAL variable a value with more significant digits to the left of the decimal point than allowed by the precision and scale of the variable.

A FIXED DECIMAL constant is written in the text of a program as one or more decimal digits with an optional decimal point. If no decimal point appears, the point is assumed to be immediately to the right of the rightmost digit. For example: 4.533, 3.14159, 003, and .0012. The precision of a FIXED DECIMAL constant is equal to the number of digits in the constant. The scale is equal to the number of digits to the right of the decimal point.

| | |
|---|---|
| Precision: | 1 < =p < =16 |
| Scale Factor: | 0 < =1 < =16, q < =p |
| Default Precision and Scale: | (7,0) |
| Alignment: | byte |
| Storage Requirement: | FLOOR((p+2)/2) bytes |
| Internal Representation: | Packed Decimal Number (See the *Programmer's Reference Manual ECLIPSE-line Computer.)* |

**Fixed Decimal**



## Float Binary(p)

A FLOAT BINARY(p) value has a sign, s, an integer exponent, e, and p binary digits which represent a hexadecimal fraction, f. A FLOAT BINARY(p) value is s*f*(16**e), where s is +1 or -1 and -64 < =e < =63. The range of FLOAT BINARY values is approximately 5.4*10**-79 to 7.2*10**75.

A FLOAT BINARY constant is written in the text of a program as a decimal number with an optional decimal point, followed by the letter E, followed by an optionally signed decimal integer that denotes a power of ten. For example, 32E-2, 1.2E+3, and -1.2E3.

FLOAT BINARY variables may be declared with any precision between 1 and 53. If the declared precision is between 1 and 21, a single precision floating point number will be allocated. The full precision of single precision floating point numbers (about 6-7 decimal digits) will be used in any computation involving the value. If the declared precision is between 22 and 53, a double precision floating point number will be allocated. The full precision of double precision floating point numbers (about 15-17 decimal digits) will be used in any computation involving the value. If a floating point variable is used in an output list, only as many bits as the declared precision of the variable specifies will be output. (See Chapter 7 on arithmetic to character string conversion.)

All FLOAT BINARY constants have a precision of 53. This can result in unexpected results if a FLOAT constant that is not exactly representable in BINARY is compared to a single precision FLOAT BINARY variable.

Example:

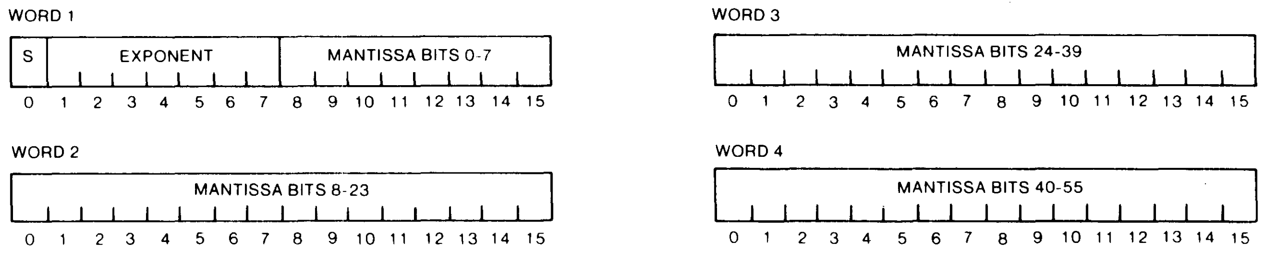
```
DECLARE A FLOAT BINARY(21);
A = 1.1E0;
IF A = 1.1E0 THEN . . .          /* This comparison will yield a result
                                 of "0"B (false) since 1.1E0 loses
                                 precision when assigned to the
                                 single-precision variable A. */
```

Integer values are exactly represented by FLOAT BINARY variables. A single precision variable can contain integer values up to ±(2**24)-1 (±16,777,216). A double-precision variable can contain integer values up to ±(2**56)-1.

If you attempt to assign to a FLOAT variable a value too large to be represented, a floating point overflow will occur and the error condition will be signalled. If you attempt to assign a value too small to be represented, the variable will be set equal to zero.

| | |
|---|---|
| Precision: | 1 < =p < =53 |
| Default Precision: | 21 |
| Alignment: | word |
| Storage Requirement: | 1 < =p < =21   2 words<br>22 < =p < =53   4 words |
| Internal Representation: | For details, see the *(Programmer's Reference Manual ECLIPSE-Line Computer).* |

**Float Binary**



## Picture

A pictured value is a FIXED DECIMAL value stored as a character-string containing decimal digits and editing characters.

PICTURE data is used to output numeric values with special characters appearing before or after the number, and in between specified digit positions.

Example:

```
DECLARE A PICTURE "$ZZZ,ZZ9V.99";
A = 1234.56;
PUT LIST (A);

output: $□□1,234.56
```

Your program may assign arithmetic values to PICTURE variables and may perform arithmetic operations on them.

A picture consists of a sequence of character codes called picture characters enclosed by quotation marks. Picture characters may be grouped into the following categories:

- 9 and V are the digit and decimal point specifiers.
- Z and * are the zero suppression characters.
- , . / and B are the insertion characters.
- $ is the dollar sign.
- + - S CR and DB are the sign characters.

You may use picture characters in various combinations. A valid picture must contain at least one picture character that specifies a digit position (9, Z, or *).

| | |
|---|---|
| 9 | specifies that the associated position in the data item is to contain a decimal digit. |
| V | specifies that a decimal point is assumed at this position in the data item. It does not specify that the actual decimal point is to be inserted. The integer and fractional parts of an assigned value are aligned on the picture character V. If the picture character V is not specified within a picture, the picture is treated as though the V were the rightmost picture character. This may cause an assigned value to be truncated. The picture character V can appear only once in a picture. The V does not occupy storage in the output representation of the picture value. |

The zero suppression picture characters specify conditional digit positions in the character string value and will cause leading zeroes to be replaced by asterisks or blanks.

| | |
|---|---|
| Z | specifies a conditional digit position and causes a leading zero in the associated data position to be replaced by a blank character. When the associated data position does not contain a leading zero, the digit in the position is not replaced by a blank character. A Z that occurs to the right of the picture character V does not cause zero suppression unless all integer and fractional digit positions contain a Z and all associated data positions contain a zero. You cannot use the picture character Z in the same picture as the picture character *, nor can it appear with a drifting picture character or to the right of the 9 picture character. |
| * | specifies a conditional digit position. It is used in the same way as the picture character Z, except that leading zeroes are replaced by asterisks. The picture character * cannot appear in the same picture as the picture character Z, nor can it appear with a drifting picture character or to the right of a 9-picture character. |

The picture characters comma (,), point (.), slash (/), and blank (B) are insertion characters. They do not indicate digit positions but are inserted between digits. Insertion characters are applicable only to the character-string value. They specify nothing about the arithmetic value of the data item. Whether or not the character is suppressed, each represents a character position in the character-string value. The comma, point, and slash are conditional insertion characters; within a string of zero suppression characters, they may be suppressed. The blank (B) is an unconditional insertion character; it always specifies that a blank is to appear in the associated position.

| | |
|---|---|
| , | causes a comma to be inserted in the associated position of the numeric character data when no zero suppression occurs. If zero suppression occurs and the previous character is an "*", the "," is replaced by "*". If zero suppression occurs and the previous character is not an "*", the "," is replaced by a blank. |

| | |
|---|---|
| . | is used the same way the comma picture character is used, except that a decimal point is assigned to the associated position. This character does not cause alignment of the integer and fractional parts of an assigned value to the indicated position. See the discussion of the picture character V. Note that if the point character precedes the V, it will be inserted only if the preceding digit position is unsuppressed. If the point character follows the V, it will be suppressed only if all digit positions are suppressed. |
| / | is used the same way the comma picture character is used, except that a slash (/) is inserted in the associated position. |
| B | specifies that a blank character will always be inserted into the associated position of the character-string value of the numeric character data. |

The picture characters S, +, and - specify signs in the numeric data. The picture character $ specifies the dollar sign in the character string value of the numeric data.

The +, -, S, and $ picture characters may be used in either a static or a drifting manner. The static use of these characters specifies that a sign, a dollar sign, or a blank always appears in the associated position.

A drifting character is specified by multiple uses of that character in a picture. The drifting character must be specified in each digit position through which it may drift. A drifting field specifies that the associated positions in the data item are to contain any significant digits in the arithmetic value being represented, preceded by the drifting character specified by the following list, preceded by enough blanks to complete the field. If the drifting field does not contain enough positions for all the significant digits, plus one for the drifting character, the program is in error.

Only one type of sign picture character can appear within a picture. A static sign or a $ (or both) may only appear on the left or the right end of a picture. A picture may contain only one picture character that drifts. A drifting picture character may not be preceded by any digit picture characters. If a picture contains a Z or an *, it may not contain any drifting picture characters.

| | |
|---|---|
| $ | specifies the dollar sign. |
| S | specifies the plus sign character (+) if the data value is > =0; otherwise, it specifies the minus sign character (-). |
| + | specifies the plus sign character (+) if the data value is > =0; otherwise, it specifies a blank. |
| - | specifies the minus sigh character (-) if the data value is < 0; otherwise, it specifies a blank. |

For example:

```
DECLARE A PICTURE "$ZZZZ9";
DECLARE B PICTURE "$$$$$9";
A = 22; B = 22;
PUT LIST (A);
PUT LIST (B);
```

```
output:     $□□□22
            □□□$22
```

The picture character pairs CR and DB specify credit and debit, respectively. Only one pair may appear in a picture. The characters CR and DB can appear only to the right of all digit positions of a picture.

| | |
|---|---|
| CR | specifies that the associated positions will contain the letters CR if the value of the data is less than zero; otherwise, the positions will contain two blanks. |
| DB | specifies that the associated positions will contain the letters DB if the value of the data is less than zero; otherwise, the position will contain two blanks. |

Additional examples of pictured data and examples of the effects of pictures on data conversion are found in Sections 4.8 and 7.9.

The precision and scale (p,q) of a PICTURE variable are not declared explicitly, but are implied by the declaration of the PICTURE, according to the following rules:

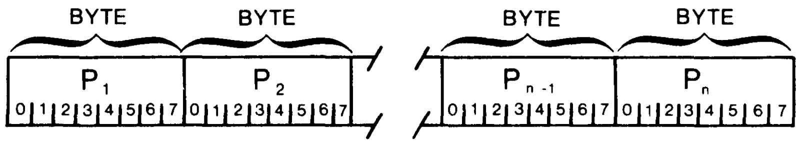
- If the picture contains only static picture characters, p is the number of Z, *, and 9 picture characters in the picture. If a V is present, q is the number of Z, *, and 9 picture characters that follow the V; otherwise, q is zero.
- If the picture contains drifting picture characters, let d be the number of drifting picture characters in the picture. The precision, p, is d-1 plus the total number of 9 picture characters. If a V is present q is the number of drifting picture characters and 9 picture characters that follow the V; otherwise, q is zero.
- The resulting values of p and q must satisfy the relationship $q <= p <= 16$ or the program is in error.

As with FIXED DECIMAL variables, it is an error to assign to a PICTURE variable values having more significant digits to the left of the decimal point than allowed by the precision scale of the variable; values which have a scale greater than the scale of the variable will be truncated.

| | |
|---|---|
| Alignment: | byte |
| Storage Requirement: | n bytes, where n is the number of picture characters, excluding any V character, in the picture. |

Internal Representation:

**Picture**



## 2.2 String Data

The string data types are: CHARACTER, ALIGNED CHARACTER, VARYING CHARACTER, BIT, and ALIGNED BIT.

There are two types of strings: character-strings and bit-strings. A bit-string value is a sequence of bits, and a character-string value is a sequence of ASCII characters. The length or *extent* of a character string is the number of characters in the string. The length or extent of a bit string is the number of bits in the string.

A bit-string value with no bits is a *null bit-string* of length zero; a character-string value with no characters is a *null character-string* of length zero.

The length of a string value is determined by the operation that produces the value and is limited only by the amount of storage available for holding the value.

The length of a string variable is determined by its declaration. The declared length of a string variable specifies the maximum length of string values that can be assigned to the variable. It also determines the amount of storage allocated for the variable.

The following descriptions give the storage requirements and alignment rules for string data.

## Character(n)

The length of a CHARACTER variable is determined when the variable is allocated and does not vary thereafter.

Any value assigned to a CHARACTER(n) variable is either truncated or padded on the right with blanks to become a string of n characters.

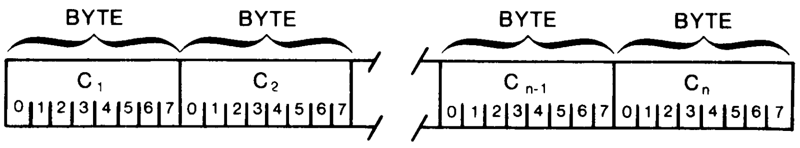
A character string constant is written in the text of a program as a sequence of ASCII characters surrounded by quotes. To include a quote character in a string, use a pair of quotes.

For example:

| | |
|---|---|
| "HE SAID, ""I DON'T KNOW""" | represents HE SAID, "I DON'T KNOW" |
| """" | represents " |
| "ABCDEFG" | represents ABCDEFG |

| | |
|---|---|
| Default Length: | 1 |
| Alignment: | byte |
| Storage Requirement: | n bytes, where n is the declared length of the variable. |
| Internal Representation: | |

**Character**



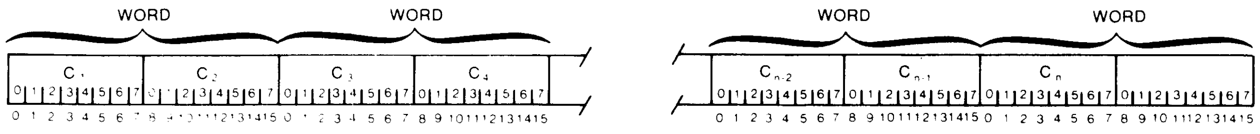
## Aligned Character(n)

The length of an ALIGNED CHARACTER variable is determined when the variable is allocated and does not vary thereafter.

Any value assigned to an ALIGNED CHARACTER(n) variable is either truncated or padded on the right with blanks to become a string of n characters.

| | |
|---|---|
| Default Length: | 1 |
| Alignment: | The aligned attribute specifies that the data is allocated on a word boundary, i.e., the alignment is word. |
| Storage Requirement: | FLOOR ((n+1)/2) words, where n is the length of the variable. |
| Internal Representation: | |

**Aligned Character**



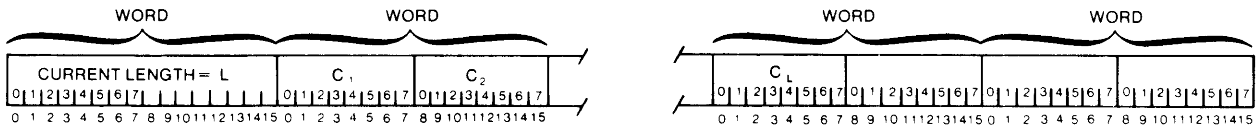
## Varying Character(n)

The varying attribute specifies that the variable represents strings of different lengths. A VARYING CHARACTER variable contains both the current length of its string value and the value. If a string value of more than n characters is assigned to a VARYING CHARACTER(n) variable, the rightmost excess characters are truncated and the current length of the variable becomes n. If a string of n or fewer than n characters is assigned to it, the string is not padded on the right, but the number of characters in the string becomes the current length of the variable.

| | |
|---|---|
| Default Length: | 1 |
| Alignment: | word |
| Storage Requirement: | FLOOR((n+1)/2 + 1) words, where n is the declared length of the variable. |

Internal Representation:

**Varying Character**



## Bit(n)

The length of a BIT variable is determined when the variable is allocated and does not vary thereafter.

Any value assigned to a BIT(n) variable is either truncated or padded on the right with zero bits to become a string of n bits.

A bit string constant is written in the text of a program as a sequence of digits, 0 through 9 and A through F, enclosed in quotation marks, immediately followed by the letter B, and optionally followed by a digit in the range 1 to 4. If this following digit is missing, then 1 is assumed. The digit indicates the number of bits in the bit string that each digit in the sequence represents.

A bit string constant is the bit string value formed by converting each of the characters contained within quotes to a series of n bits where n is 1 for B or B1 format, 2 for B2 format, 3 for B3 format, and 4 for B4 format according to the following table:

| Character | B or B1 | B2 | B3 | B4 |
|---|---|---|---|---|
| 0 | 0 | 00 | 000 | 0000 |
| 1 | 1 | 01 | 001 | 0001 |
| 2 | invalid | 10 | 010 | 0010 |
| 3 | | 11 | 011 | 0011 |
| 4 | | invalid | 100 | 0100 |
| 5 | | | 101 | 0101 |
| 6 | | | 110 | 0110 |
| 7 | | | 111 | 0111 |
| 8 | | | invalid | 1000 |
| 9 | | | | 1001 |
| A | | | | 1010 |
| B | | | | 1011 |
| C | | | | 1100 |
| D | | | | 1101 |
| E | | | | 1110 |
| F | | | | 1111 |

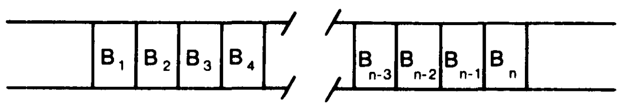
For example, each pair of constants in the following list, represents the same value:

| | |
|---|---|
| "01011111"B1 | "01011111"B |
| "1"B1 | "1"B |
| "012"B2 | "000110"B |
| "011011110010"B1 | "011011110010"B |
| "123302"B2 | "011011110010"B |
| "3362"B3 | "011011110010"B |
| "6F2"B4 | "011011110010"B |

| | |
|---|---|
| Default Length: | 1 |
| Alignment: | bit |
| Storage Requirement: | n bits, where n is the declared length of the variable. |

Internal Representation:

**Bit**



## Aligned Bit(n)

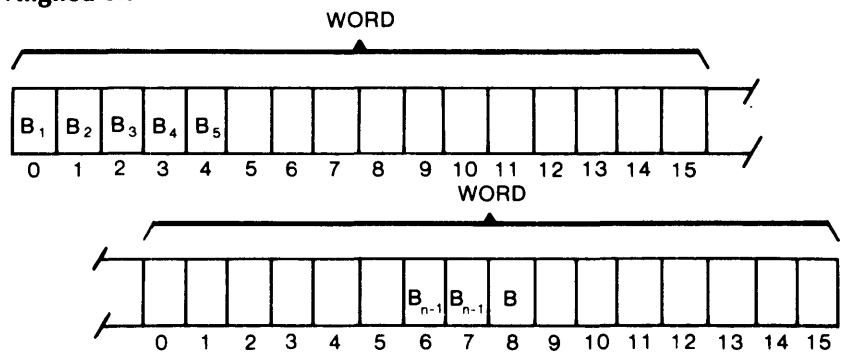
The length of the ALIGNED BIT variable is determined when the variable is allocated and does not vary thereafter.

Any value assigned to an ALIGNED BIT(n) variable is either truncated or padded on the right with zero bits to become a string of n bits.

| | |
|---|---|
| Default Length: | 1 |
| Alignment: | word |
| Storage Requirement: | FLOOR((n+15)/16) words, where n is the declared length of the variable. |

Internal Representation:

**Aligned Bit**



## 2.3 Pointer Data

A *pointer* identifies a generation of storage of a variable (see Section 3.1). The value of a pointer variable is effectively a memory address. In other words, a pointer indicates exactly where in the computer's memory a variable is located.

A pointer can identify the storage of any word or byte aligned variable, but not of any bit aligned variable. A pointer that identifies a word aligned variable contains the word-address of the variable, and is called a *word-pointer.* A pointer that identifies a byte aligned variable contains a byte address, and is called a *byte-pointer.*

Pointer values can be obtained by the use of the ADDR built-in function, and from the SET option of the ALLOCATE statement. The NULL built-in function is used to obtain a *null pointer,* which is a unique value that does not identify a generation of storage.

You may assign pointer values to pointer variables using the assignment statement, and pointer values may be passed as arguments to procedures. However, no operations may be performed on a pointer value.

You use a pointer to distinguish between multiple generations of storage of a variable, and to qualify a reference to a BASED variable.

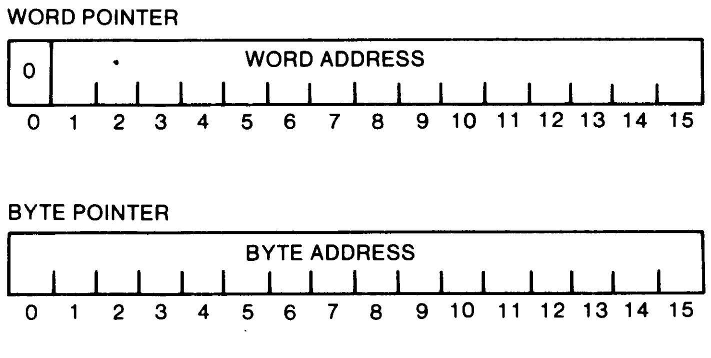
A pointer loses its validity when the generation of storage it identifies is freed. It is an error to use an invalid or null pointer to identify any allocation of a BASED variable.

A pointer derived from either allocating or taking ADDR of a variable X can only be used to access data whose type is the same type as X, with those exceptions given in Section 3.5 on storage sharing. Violation of this constraint will result in errors during program execution.

| | |
|---|---|
| Alignment: | word |
| Storage Requirement: | 1 word |

Internal Representation:

**Pointer**



## 2.4 Label Data

A label constant is an identifier written as a prefix to any statement other than a PROCEDURE statement, ENTRY statement, or FORMAT statement. A colon connects the label to the statement.

```
AB1: X = Y+Z; /* AB1 is the statement label */
```

The value of a label variable is derived by assigning a label constant to it, by passing a label constant as an argument, or by returning a label constant as the value of a function.

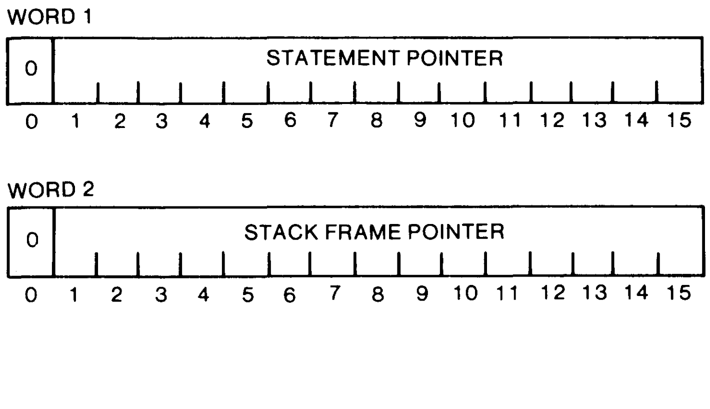
```
DECLARE VARLBL LABEL;                    /* VARLBL is a label variable */
.
.
.
AB1: X = Y+Z;                            /* AB1 is a label constant */
.
.
.
AB2: X = Y;                              /* AB2 is a label constant */
.
.
.
VARLBL = AB1;                            /* the label constant AB1 is
                                         assigned to the label variable VARLBL */
.
.
.
GOTO VARLBL;                             /* causes a transfer to AB1 */
.
.
.
```

*Figure 2-1. Label Data*

| | |
|---|---|
| Alignment: | word |
| Storage Requirement: | 2 words. The first word is a word pointer to the statement associated with the label. The second word is a word pointer to the stack frame that indicates the environment associated with the label. |

Internal Representation:

**Label**



## 2.5 Entry Data

An entry constant identifies an entry point to a procedure or an entry statement within a procedure. An entry constant is an identifier written as a label of an ENTRY statement or a PROCEDURE statement.

There are external entry constants and internal entry constants. An *external entry constant* identifies an entry point to an external procedure. An *internal entry constant* identifies an entry point to a nested procedure.

The value of an entry variable is derived by assigning an entry constant to it, by passing an entry constant as an argument, or returning an entry constant as the value of a function.

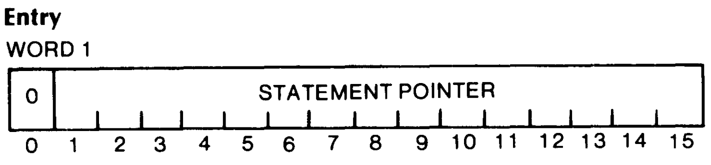
```
P:  PROCEDURE;
    DECLARE W ENTRY VARIABLE;
    .
    .
    .
E:  ENTRY;
    .
    .
    W = S;               /* the entry constant S is assigned to the
    .                    entry variable W */
    CALL W;              /* call S */
    .
    .
S:  PROCEDURE;
    .
    .
F:  ENTRY;
    .
    .
    END;                 /* S */
    END;                 /* P */
```

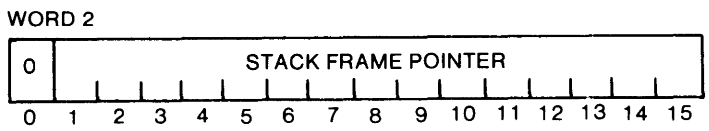
*Figure 2-2. Entry Data*

P and E are external entry constants. S and F are internal entry constants. W is an entry variable.

| | |
|---|---|
| Alignment: | word |
| Storage Requirement: | 2 words. The first word is a word pointer to the statement associated with the entry constant. The second word is a word pointer to the stack frame which indicates the environment associated with the correct block activation of the procedure. |

Internal Representation:





## 2.6 File Data

File data is used to identify the data set (file) to be used by input and output statements (see Chapter 8).

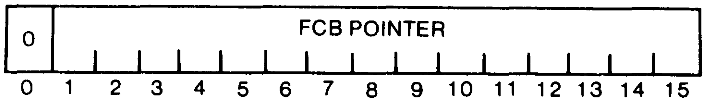
The value of a file constant is the address of a specific file-control block. A program has as many file-control blocks as it has file constants. See Chapter 8 for a detailed explanation of the file-control block.

The value of a file variable is derived by assigning a file constant to it, by passing a file constant as an argument, or by returning a file constant as the value of a function.

| | |
|---|---|
| Alignment: | word |
| Storage Requirement: | 1 word |

Internal Representation:

**File**



## 2.7 Aggregates of Data

Each of the data types discussed in the previous sections may be single data items or they may be grouped together to form arrays or structures.

### Arrays

An *array* is a named, ordered collection of values all having the same attributes. You can reference an entire array by using its name, or an individual item in an array, or *element,* by specifying its relative position in the array by means of subscripts.

An array can have from one to eight *dimensions;* the number of dimensions is the number of subscripts necessary to reference an element of the array. The *bounds* of a dimension is the range of values a subscript for that dimension can have.

Example:

```
DECLARE AR(1:5,1:3) FIXED BINARY;
```

AR is a two-dimensional array. The *lower bound* of each dimension is 1. The *upper bound* of the first dimension is 5 and of the second dimension is 3. AR(3,1) is a reference to a single element of the array. AR(I,J) is a legal reference to an array element if 1 < =I < =5 and 1 < =J < =3. A subscript can be any integer-valued expression.

If only one bound is given for a dimension in the declaration of an array, it is the upper bound, and the lower bound is assumed to be 1. The lower and upper bounds of a dimension may be any positive or negative integer or zero, but the upper bound must be greater than or equal to the lower bound. The *extent* of a dimension is the number of different values its subscript can have, equal to its upper bound - lower bound + 1. The total number of elements in an array is equal to the product of the extents of its dimensions.

The elements of an array are stored in memory as an ordered sequence. To reference the elements of an array in the order in which they are stored, the rightmost subscript should vary most rapidly and the leftmost subscript least rapidly. This ordering is called *row-major order.*

Example:

```
DECLARE ARY(0:2,2) FIXED BINARY;
```

The elements of ARY are stored in the following order:

```
ARY(0,1) ARY(0,2) ARY(1,1) ARY(1,2) ARY(2,1) ARY(2,2)
```

The storage requirement of an array is equal to the number of elements in the array times the storage requirement of one of its elements. The alignment of an array is the alignment of its elements.

## Structures

A *structure* is a hierarchically ordered set of values. The values do not necessarily have the same data types. A structure can contain scalar or array variables of any data type, and also can contain nested structures, or *substructures*. The data items and substructures which make up a structure are *members* of the structure. The outermost structure is a *major structure*. Substructures can be called *minor structures*.

The organization of a structure is specified in a DECLARE statement through the use of level numbers. A major structure is given a name and must be declared with the level number 1. Substructures and other members of the major structure are declared with level numbers that are integer constants greater than 1. Members of substructures must be declared with level numbers greater than the level number of the substructure. Members of a structure are referenced by structure-qualified references as described in Section 5.3.

For example, the items of a directory could be declared as follows:

```
    DECLARE 1  DIRECTORY,
               2 NAME,
                     3 LAST CHARCTER(20),
                     3 FIRST CHARACTER(10),
                     3 MIDDLE_INITIAL CHARACTER(1),
               2 ADDRESS,
                     3 NUMBER FIXED DECIMAL(6),
                     3 STREET CHARACTER(20),
                     3 CITY CHARACTER(20),
                     3 STATE CHARACTER(20),
                     3 ZIP FIXED DECIMAL(5),
               2 TELEPHONE_NUMBER FIXED DECIMAL(10);
```

DIRECTORY is the major structure name. NAME is a substructure name whose members are LAST, FIRST, and MIDDLE_INITIAL. ADDRESS is a substructure name whose members are NUMBER, STREET, CITY, STATE, and ZIP. TELEPHONE_NUMBER is a member of the major structure. See Section 5.3 which discusses structure qualified references.

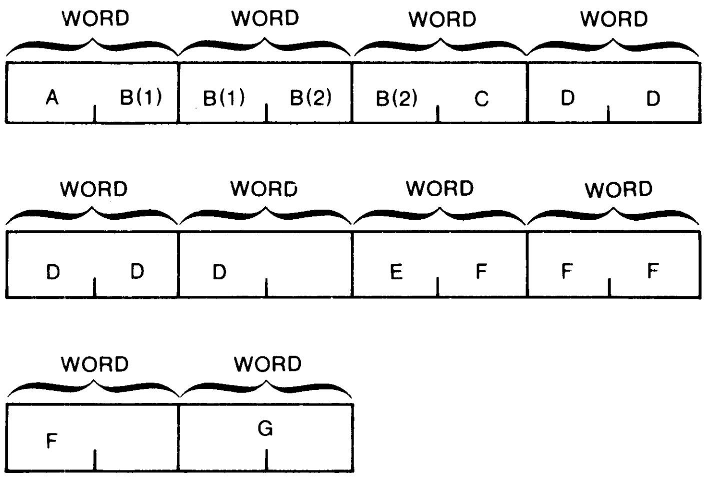
An alignment of a major or minor structure is the maximum of the alignments of its members, with word alignment considered to be the largest. The members of a structure are stored in the order in which they are declared. The storage requirement of a structure is equal to the sum of the storage requirements of its members.

For example:

```
DECLARE 1 S,
           2 S1,
               3 A CHARACTER(1),
               3 B(2) CHARACTER(2),
           2 S2,
               3 C CHARACTER(1),
               3 D CHARACTER(5),
           2 S3,
               3 E CHARACTER(1),
               3 F CHARACTER(4),
               3 G FIXED BINARY;
```

Substructures S1 and S2 are byte aligned. Substructure S3 is word aligned since it contains a word aligned item. S is word aligned since it contains a word aligned item (S3).

**Structures**



## Arrays of Structures

An array of structures is an array whose elements are structures.

For example:

```
DECLARE 1 BUDGET(12),
            2 INCOME FIXED DECIMAL(5,2),
            2 OUTGO,
                 3 ELECTRICITY FIXED DECIMAL(5,2),
                 3 TELEPHONE FIXED DECIMAL(5,2),
                 3 RENT FIXED DECIMAL(5,2);
```

The array BUDGET is an array of 12 structures. Budget data for the month of February could be specified by BUDGET(2) or in particular, BUDGET(2).RENT and BUDGET(2).INCOME. See Section 5.3.

## Connected and Unconnected Arrays

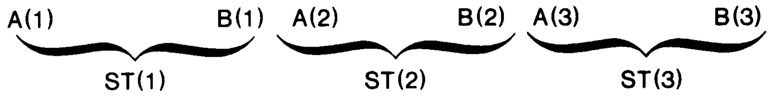
A *connected array* is an array whose elements are not separated from one another in storage by other values. All arrays, except members of dimensioned structures, are connected arrays.

An *unconnected array* is an array whose elements are separated from one another in storage by other values. Members of dimensioned structures are unconnected arrays.

For example:

```
DECLARE 1 ST(3),
            2 A FIXED BINARY(3),
            2 B FIXED BINARY(3);
```

Storage is allocated in the following way:



In this example, ST is a connected array because storage for the elements of ST does not contain gaps occupied by other data. However, A and B are unconnected arrays because storage for their elements contain gaps occupied by elements of another array.

## 2.8 Summary of Data Alignment and Size

| Type | Alignment | Size |
|---|---|---|
| FIXED BINARY | word | 1 word |
| FIXED DECIMAL | byte | FLOOR((p+2)/2)bytes |
| FLOAT BINARY(p),1 <=p<=21 | word | 2 words |
| FLOAT BINARY(p), 22<=p<=53 | word | 4 words |
| PICTURE | byte | n bytes |
| CHARACTER(n) | byte | n bytes |
| ALIGNED CHARACTER(n) | word | FLOOR((n+1)/2 words |
| VARYING CHARACTER(n) | word | FLOOR((n+1)/2+1)words |
| BIT(n) | bit | n bits |
| ALIGNED BIT(n) | word | FLOOR((n+15)/16) word |
| POINTER | word | 1 word |
| LABEL | word | 2 words |
| ENTRY | word | 2 words |
| FILE | word | 1 word |
| array | the same as one of its elements | the storage requirement of one element times the number of elements. |
| structure | the maximum of its members | the sum of its members |

NOTE: The alignment of data is determined strictly by the data's type and is not affected by its relative position within a structure or array. For example, the fact that a variable's storage happens to start of a word boundary does not make the variable word aligned.

All elementary data items, arrays, and major structures begin on a word boundary regardless of their data type or alignment. Elementary data items, arrays, and major structures are referred to as level-one data items.

End of Chapter

# Chapter 3
# Storage Classes

## 3.1 Allocation of Storage

A *generation of storage* is a sequence of bits of sufficient length to represent all of the values within the range of a variable's data type.

When storage has been associated with a variable, the variable is said to be allocated. The manner in which storage is allocated for a variable is determined by the *storage class* of the variable. There are five storage classes: AUTOMATIC, INTERNAL STATIC, EXTERNAL STATIC, BASED, and DEFINED.

Storage class attributes may be declared explicitly for scalar, array, and structure variables. If no storage class is explicitly declared for a variable and the variable is not a parameter, the variable is given the AUTOMATIC storage class. If a variable is an array or structure variable the storage class declared for that variable applies to all of the elements in the array or structure.

An *extent expression* is an expression in a DECLARE statement which determines the bounds of arrays and the length of string variables (see Chapter 4). The type of expression allowed as an extent expression depends on the storage class of the variable. Parameters may be declared with an asterisk as an extent expression (see Section 5.6).

The allocation of a generation of storage for a variable consists of performing the following steps:

- Evaluate each extent expression specified in the variable's declaration.
- Determine the amount of storage required by examining the data type and ALIGNMENT attributes together with the evaluated extents from the previous step.
- Obtain a generation of storage of sufficient size. If the variable being allocated is not BASED, associate the newly allocated generation with the name of the variable. If the variable being allocated is BASED, assign a pointer that identifies the newly allocated generation to the pointer variable given by the SET option of the ALLOCATE statement that caused this allocation to occur.
- If the variable being allocated is BASED, assign the evaluated left part of each REFER option to the variable identified by the REFER option's right part.
- Evaluate each INITIAL attribute specified in the variable's declaration and assign initial values to the newly allocated generation. Note that the INITIAL attribute can only be specified for INTERNAL STATIC or EXTERNAL STATIC storage, and that such storage is allocated prior to program execution.

## 3.2 Automatic Storage

A generation of storage is allocated for each automatic variable declared in a given block each time the block is activated. The variable remains allocated as long as the block remains active. Recursive activation of a block has the effect of stacking generations of the block's automatic variables.

A unit of storage called a *stack frame* is allocated for each block activation. This unit of storage contains information needed by control in order to execute the statements in the block and is the place where all automatic variables declared in the block are allocated. Label and entry data contain as part of their value a pointer to a stack frame.

When a procedure is invoked recursively, a new stack frame is created for each activation. The stack frame contains a new copy of all automatic data declared in the procedure. Any reference to an automatic variable is a reference to the copy associated with the most recent activation of the block. Similarly, a reference to a parameter, defined variable, label constant, or entry constant within the block is a reference to the most recent activation of the block.

The extent expressions of an automatic variable can contain expressions whose values are computable upon block activation. A value is computable upon block activation if it can be computed without referencing any automatic or defined variable declared in the block.

The extent expressions are evaluated and stored in the stack frame. Subsequent references to the generation of the automatic variable use these evaluated extents.

Automatic variables become undefined when the block in which they are declared becomes inactive. It is an error to use a pointer which has been assigned the address of a variable which has become undefined through the deactivation of the block in which it is declared.

For example:

```
A:              PROC;
DECLARE PT POINTER;
DECLARE CH1 CHARACTER(10) BASED(PT);
DECLARE CH2 CHARACTER(10);
                BEGIN;
                DECLARE CH3 CHARACTER(10);
                CH3 = "START";
                PT = ADDR(CH3);
                END;
CH2 = PT->CH1;
```

The last statement is in error, since the generation of storage identified by PT becomes undefined when the begin block is terminated.

All variables that have not been explicitly declared with a storage class attribute are assumed to have the AUTOMATIC attribute.

## 3.3 STATIC Storage

A single generation of storage is allocated for each static variable before the execution of the program begins and remains allocated for the duration of the program. Because static variables are allocated before the execution of the program, they must have constant extent expressions or initial attributes.

There are two kinds of static variables: INTERNAL STATIC and EXTERNAL STATIC.

INTERNAL STATIC variables are static variables whose scope is internal; i.e., the variable name is known only in the block in which it is declared and in all contained blocks except blocks where it is redeclared.

For example:

```
A:      PROCEDURE;
        DECLARE X FIXED DECIMAL INTERNAL STATIC INITIAL(0);
        .              /* X is known within A and B*/
        .
        .
        .
        B:             PROCEDURE;
                       .
                       .
                       .
                       X = X+1;           /* count the number of
                       .                  times B is executed*/
                       END;               /* B */
        .
        .
        .
        .
        END;                              /* A */
```

The use of the INTERNAL STATIC INITIAL attributes causes storage to be allocated for X and the value of the variable X to be set before execution of the program and to remain set for the next invocation of the procedure A.

EXTERNAL STATIC variables are static variables whose scope is externa!; i.e., the variable name is known in all blocks in which the name is declared with the EXTERNAL attribute and in all contained blocks except those blocks in which it is redeclared with any other attribute. See Chapter 4 which discusses the scope of a declaration.

For example:

```
C:      PROCEDURE;
        DECLARE Z FIXED BINARY(15) EXTERNAL STATIC;
        .
        .
        .
D:                     PROCEDURE;
                       DECLARE Z FIXED DECIMAL(3,2);
                       .
                       .
                       .
                       END; /* D */
        .
        .
        .
E:                     PROCEDURE;
                       DECLARE 7 FIXED BINARY(15) EXTERNAL STATIC;
                       .
                       END; /* E */
        END /* C */

G:      PROCEDURE;
        DECLARE Z FIXED BINARY(15) EXTERNAL STATIC;
        .
        .
        .
        END; /* G */
```

The identifier Z refers to the same variable in procedures C, E, and G. A separate variable named Z is known in procedure D.

## 3.4 BASED Storage

A based variable is a description of a generation of storage, but no generation is ever directly associated with the name of the based variable. The specific generation of storage accessed by a reference to a based variable is specified by a pointer. The pointer identifies a generation of storage whose attributes are described by the based variable.

A generation of storage is allocated for a based variable only upon execution of an ALLOCATE statement specifying that variable. The variable remains allocated for the duration of the program or until the execution of a FREE statement specifying that variable. You control the allocation and freeing of storage for a based variable.

Whenever a based variable is allocated, a pointer is set to a value which is effectively the address of the allocation. A based variable can be reallocated without first being freed. The earlier allocation is not lost; however, it is your responsibility to save the value of a pointer associated with the previous allocation. By explicitly specifying a pointer when a based variable is referenced, you can distinguish between distinct allocations of one based variable.

For example:

```
DECLARE (P,X) POINTER;
DECLARE A FIXED BASED;
.
.
.
ALLOCATE A SET(P);           /*P is set to identify a generation of
                             storage for A*/
.
.
.
ALLOCATE A SET(X);           /*X is set to identify a second
                             generation of storage for A */
.
.
.
```

The first allocation of A can be referenced by P- > A and the second by X- > A.

There are two ways to derive the value of a pointer. If the value of the pointer is derived from the execution of an ALLOCATE statement the generation is called an explicitly allocated based generation.

For example:

```
DECLARE X FIXED BASED;
.
.
.
ALLOCATE X SET(P);           /* P is set to the location of X in
                             storage*/
```

Multiple copies of a BASED structure can be allocated and chained together by means of pointers that are declared as part of the structure.

For example:

```
DECLARE (P,Q) POINTER;
DECLARE 1 LIST BASED(P),
              2 INFORMATION
                    3 INFO1 FIXED,
                    .
                    .
                    .
                    3 INFO10 FIXED,
              2 PREVIOUS POINTER,
              2 NEXT POINTER;
.
.
.
Q = NULL;
DO I = 1 TO N;
ALLOCATE LIST SET(P);
IF I>1 THEN Q->LIST. NEXT = P;
LIST.PREVIOUS = Q;
LIST.NEXT = NULL;
Q = P;
END;
```

This creates a chain of N structures called LIST, with each copy containing a pointer to the previous and next items in the chain.

If the value of the pointer is derived from the evaluation of an ADDR built-in function the generation is called an equivalenced based generation.

For example:

```
DECLARE Y FIXED DECIMAL(3);
DECLARE X FIXED DECIMAL(3) BASED;
.
.
.
P = ADDR(Y);              /*P is set to the address of Y*/
.
.
.
P->X = 2;                           /*assigns 2 to Y*/
.
.
.
```

This technique is used to share storage among 2 or more variables. See the section in this chapter which describes storage sharing.

The extent expressions in the declaration of a based variable are not evaluated upon activation of the block which contains the declaration of the based variable. The extent expressions can therefore contain automatic and defined variables which are declared in the same block. These variables must be assigned a value before the first allocation of the based variable.

The extent expressions are evaluated for each allocation and again for each reference to the based variable. Consequently, if the values of the extent expressions change during execution after the variable is allocated, so will the size of the based variable. It is your responsibility to ensure that these extent expressions accurately describe the extents of the generation referenced by the based variable.

Example:

```
DECLARE N FIXED;
DECLARE X CHARACTER(N) BASED;
.
.
.
N = 25;
.
.
.
ALLOCATE X SET(P);              /* N is evaluated here and determines the
.                               amount of storage allocated*/
.
.
IF P->X = Y THEN ...            /*N is evaluated again and must be 25 or less*/
.
.
.
```

Example:

```
BAS:PROC;
DECLARE P POINTER;
DECLARE (I,J) FIXED;
DECLARE 1 A BASED(P),
            2 B(I) FIXED,
            2 C(I) FIXED;

I=10;
ALLOCATE A SET(P);
DO J=1 TO 10;
B(J)=J;C(J)=J;
END;
PUT SKIP LIST (B(1),B(I),C(1),C(I));
I=5;
PUT SKIP LIST (B(1),B(I),C(1),C(I));
END;
```

Output:

```
1        10       1        10
1        5        6        10
```

When the second line is output, the extent expression I is equal to 5. The array B fills only 5 words. The array C fills the next 5 words, previously occupied by B(6) through B(10).

## The REFER Option

The REFER option can only be used in the extent expressions of members of based structures. Such structures are called self-defined structures. A *self-defined structure* is one which contains within itself size information about its own fields.

The REFER option is used in the declaration of a based structure to specify that, on allocation of the structure, the value of an expression is to be assigned to a member of the structure, and that this value will be the extent expression of an ensuing member of the same allocation of the structure.

The form of the REFER option is:

```
expression REFER(reference)
```

where the expression must produce a FIXED BINARY value. The reference cannot be pointer qualified or subscripted. The reference must be a FIXED BINARY member of the structure and must be declared earlier in the structure. Note that the reference cannot identify a member of a dimensioned structure or the element of an array.

For example:

```
DECLARE 1 ABC BASED,
            2 K FIXED,
            2 A CHARACTER(N REFER (ABC.K));
.
.
.
ALLOCATE ABC SET(P);
```

To determine the amount of storage required, N is evaluated at the execution of the ALLOCATE statement. The storage is allocated and N is assigned to P- > ABC.K. N is used only on allocation. Further references to P- > ABC.A will use ABC.K as the size of A.

### Implicit Pointer Qualification

If a based variable is declared with a BASED attribute that contains a reference to a pointer variable or pointer valued function, any reference to the based variable that is not explicitly qualified by a pointer is implicitly qualified by the pointer given in the BASED attribute.

For example:

```
DECLARE X FIXED BASED(P);
.
.
.
Y = X;
Z = Q->X;
```

In this example, the unqualified reference to X is equivalent to P- > X.

## 3.5 Storage Sharing

There are three ways for two or more variables to share a generation of storage: storage sharing by parameters, storage sharing by based variables, and storage sharing by defined variables.

### Storage Sharing by Parameters

When a variable is passed by reference to a parameter, the variable and the parameter refer to the same generation of storage and therefore share that generation. Chapter 5 discusses argument passing by reference.

For example:

```
CALL FUNC(X);
.
.
.
FUNC: PROC(Y);
```

During the block activation of FUNC caused by the execution of CALL FUNC(X), X, and Y refer to the same generation of storage.

## Storage Sharing by Based Variables

Since the pointer value identifying a generation of a variable in any storage class can be derived by the use of the ADDR built-in function, it is possible for a based variable to be effectively equivalenced to a generation in any storage class.

For example:

```
DECLARE A FIXED AUTOMATIC;
DECLARE B FIXED BASED;
P = ADDR(A);
P->B = 5;
```

The value of A after execution of the last assignment statement is five.

It is also possible for several based variables to be referenced using the same pointer, thus effectively equivalencing all of those based variables to the same generation of storage.

```
P->X                    P->Y     P->Z
```

In this example, the based variables X, Y, and Z are equivalenced to the generation of storage identified by P.

Storage sharing by based variables is restricted by the following two constraints and can only be done under the criteria for storage sharing listed below. The two constraints are:

1. A based variable cannot access the storage of an unconnected array. See Section 2.7 for a discussion of unconnected arrays.

2. A based variable cannot access the storage of a parameter, except during the block activation to which the storage was passed as an argument. For example, it is an error to take the ADDR of a parameter and assign the resulting pointer to static storage and, in another block activation, use the pointer value.

## Storage Sharing by Defined Variables

The purpose of the DEFINED attribute is to map a defined variable onto a generation of storage of another variable. Since a defined variable is associated with the generation of storage identified by its reference, it is never allocated. When specified for a structure, it maps the entire structure onto the generation of storage identified by the reference.

The extent expressions of a defined variable are evaluated upon block activation and saved in the stack frame. Therefore, they are subject to the rules for extent expressions of AUTOMATIC variables. See Chapter 3.

The variable identified by the reference cannot be a defined variable, a bit aligned variable, or a named constant.

The DEFINED attribute cannot be specified for members of structures.

For example:

```
DECLARE A(100) FIXED;
DECLARE B(10,10) FIXED DEFINED(A); /*B uses the same storage as A */
```

## Criteria of Storage Sharing

All variables that are to share the same storage by any of the above methods must satisfy one of the following three criteria:

1. Storage can be shared by two or more variables if the data type, alignment, and extents of the variables are identical. A variable declared with the PICTURE attribute can share storage only with variables declared with identical pictures. Note that this definition permits a scalar variable to share storage with an element of an array or with a member of a structure.

   This is the only way that storage can be shared by parameters and arguments. It can also be used to share storage by BASED or DEFINED variables.

   For example:

   ```
   DECLARE A(10) FIXED DECIMAL(3);
   DECLARE B FIXED DECIMAL(3) BASED;
   P = ADDR(A(5));
   ```

   P->B is a valid reference to A(5).

   ```
   DECLARE A(10) FIXED DECIMAL(6);
   DECLARE B FIXED DECIMAL(6) DEFINED(A(2));
   ```

   B is a valid reference to A(2).

   If the shared generation is a structure, the structuring of all of the variables that are used to reference the generation must be identical, and the data types, alignment, and extents of all corresponding members of the structure variables must be identical.

   For example:

   ```
   DECLARE 1 CUSTOMER(20),
                     2 ACCNO FIXED DECIMAL(6),
                     2 NAME CHARACTER(20),
                     2 ADDRESS CHARACTER(20),
                     2 BALANCE FIXED DECIMAL(3,2);

   DECLARE 1 C BASED,
                     2 AN FIXED DECIMAL(6),
                     2 N CHARACTER(20),
                     2 AD CHARACTER(20),
                     2 B FIXED DECIMAL(3,2);

   P = ADDR(CUSTOMER(3));
   ```

   P->C is a valid reference to CUSTOMER(3), and P->C.B is a valid reference to CUSTOMER(3).BALANCE.

   ```
   DECLARE 1 CUS DEFINED CUSTOMER(3),
                     2 ACT FIXED DECIMAL(6),
                     2 NME CHARACTER(20),
                     2 ADD CHARACTER(20),
                     2 BAL FIXED DECIMAL(3,2);
   ```

   CUS is a valid reference to CUSTOMER(3), and CUS.BAL is a valid reference to CUSTOMER(3).BALANCE.

2. Storage can be shared by two or more variables if the variables consist entirely of CHARACTER data or entirely of BIT data. The variable declarations cannot contain the ASSIGNED or VARYING attributes. Note that this definition permits the extents and structuring of the variables to differ. In this case pictured data is considered to be CHARACTER data. This can be used to share storage by BASED variables or DEFINED variables, but not by parameters.

   For example:

```
DECLARE X(5) CHARACTER(1);
DECLARE Y CHARACTER(5) BASED;
P = ADDR(X);
P->Y = "LMNOP";          /* the first element of the array
                         X is L and the fifth is P */
```

   or

```
DECLARE A CHARACTER(8);
DECLARE 1 B DEFINED(A),
                2 C CHARACTER(4),
                2 D CHARACTER(4);
```

   B.C is a valid reference to the first four characters of A and B.D is a valid reference to the last four characters of A.

3. Storage can be shared by two or more variables providing that all the variables are structures which have identical data types, alignment attributes, extents, and structuring from top to bottom up to and including all members contained within level two of the item being referenced. If any part of a level two structure is to be shared, all of it must be shared.

   This can be used to share storage by BASED variables, but not by parameters or DEFINED variables.

   For example:

```
DECLARE 1 S,      DECLARE 1 T BASED,      DECLARE 1 U BASED,
           2 A,                  2 A,                       2 A,
           2 B,                  2 B,                       2 B,
           3 C,                  3 C,                       3 C;
           3 D,                  3 D;
           2 E,
           2 F,
           2...;
P = ADDR(S);
```

   A reference to P->T.B.C is a valid reference to S.B.C, but a reference to P->U.B.C is not a valid reference to S.B.C because the declaration of U does not describe all of the level two substructure of S.B.

A program which violates the above constraints by defining a variable X on a variable that has a different data type than X may do so only if it is compiled without optimization, and if the alignment of X and the alignment of the referenced variable are the same.

Similarly, a program which derives the value of a pointer either by allocating or taking ADDR of a variable X, and uses this pointer to reference a variable whose type is different from X, may do so only if it is compiled without optimization, and if the alignment of X and the alignment of the referenced variable are the same.

However, programs which do violate the above constraints in this way are technically invalid PL/I programs and cannot be guaranteed to execute correctly in future implementations of PL/I.

See Section 2.8 for a table of data alignment rules.

End of Chapter

# Chapter 4 Declarations

An identifier may be used as a keyword or as the declared name of a variable or named constant. (A *named constant* is a constant that is not a literal constant. ENTRY, LABEL, and FILE constants are the types of named constants.) All names, except the names of built-in functions and pseudo-variables, must be declared in a DECLARE statement or as a label.

Label constants and entry constants are declared automatically by their use in a PL/I program. File constants and all variable names must be declared in a DECLARE statement. Also, any external procedure (other than the containing one) which is referenced in a program must be declared in a DECLARE statement.

The DECLARE statement is used to assign various characteristics to each declared name. These characteristics are called attributes. Examples of attributes are the data type of a variable (see Chapter 2) and the storage class of variables (see Chapter 3).

## 4.1 The Scope of a Declaration

The region of your program in which a declared name is known is called the scope of the declaration or scope of the name. The scope of a declaration includes the entire block in which the declaration occurs, plus any contained blocks in which the name being declared is not redeclared to mean something else.

For example:

```
A:   PROC;
     DECLARE X FIXED BINARY;
B:              PROC;
                END;              /* B */
C:              PROC;
                DECLARE X FIXED DECIMAL;
D:                          PROC;
                            END;  /* D */
                      END;        /* C */
                END;              /* A */
```

The scope of the first X is blocks A and B. The scope of the second X is blocks C and D.

A declaration that violates either of the following constraints is a multiple declaration and is in error.

- A name cannot be declared more than once in a given block except as the name of a member of a structure.

  For example;

  ```
  DECLARE 1 A,
          2 B FIXED DECIMAL(3);
  DECLARE B CHARACTER(20);
  ```

  B is a valid reference to a character string of length 20 and A.B is a structure qualified reference to a FIXED DECIMAL variable. See Chapter 5 which discusses structure qualified references.

- Two members at the same level of a structure cannot have the same name, unless they are members of different substructures.

### Internal Scope

All names, except those declared with the EXTERNAL attribute have INTERNAL scope; i.e., the name is known only in the block in which it is declared and in all contained blocks, except those blocks where it is redeclared. The INTERNAL attribute can be explicitly declared only for STATIC variables. See Section 3.3 which discusses STATIC STORAGE.

### External Scope

A declaration containing the EXTERNAL attribute is the declaration of a name whose scope is external; i.e., the name is known in all blocks in which it is declared with the EXTERNAL attribute and in all contained blocks except those blocks in which it is redeclared with internal scope. A single generation of storage is used by all blocks which reference an external data item.

Entry constants and file constants have external scope as do static variables that are explicitly declared with the EXTERNAL attribute. The EXTERNAL attribute can be explicitly declared only for STATIC variables. The compiler supplies the EXTERNAL attribute to all declarations of file constants and external entry constants. See Section 3.3 which discusses STATIC STORAGE.

## 4.2 Declaration of Labels

A declaration is established for each name that appears as a label in your program. Names that are used as labels do not have to be declared in a DECLARE statement.

For each name used as a label of an ENTRY statement, PROCEDURE STATEMENT, or BEGIN statement, a declaration is established in the block that immediately contains the procedure or begin block. This allows procedures which are immediately contained in the same block to call each other.

For example:

```
A:          PROC;

B:                      PROC;
                        CALL C;
                        END;

C:                      PROC;
                        .
                        .
                        .
                        END;
            END;
```

The declaration of B and C are established in A. Therefore, procedure C can be called from within B.

The declarations produced by labels on the ENTRY statements or PROCEDURE statement of an external procedure are established in an imaginary outer block that contains the external procedure. These labels are external entry constants. Labels on an internal procedure or entry statement are internal entry constants. See the section on ENTRY in Chapter 2.

A declaration is established in the block which immediately contains the label for all labels other than those on BEGIN, PROCEDURE, or ENTRY constants.

The label on any statement other than a PROCEDURE, ENTRY, or FORMAT statement (but including the BEGIN statement), constitutes a declaration of a label constant. See the section on LABEL data in Chapter 2.

The label on a FORMAT statement is called a format constant. Its only use is in the R format of a format-list. See Section 9.11.

A label is not allowed on a DECLARE statement, the THEN or ELSE clause of an IF statement, or a BEGIN statement which is the start of an on-unit.

For example:

```
A:      PROCEDURE;
        .
B:      .
        .
        C:      PROCEDURE;
                .
                .
        D:      .
        E:      ENTRY;
                .
                .
        F:      .
        G:      END;
        .
                .
H:      .
I:      END;
```

The names B,C,E,H,I are declared in PROCEDURE A, and since they are not redeclared in PROCEDURE C, their scope includes both A and C. The names D,F,G are declared in PROCEDURE C and their scope is PROCEDURE C. The name A is declared in an imaginary outer block and its scope includes both PROCEDURE A and PROCEDURE C.

## Subscripted Labels

A label attached to any statement other than a PROCEDURE, ENTRY, or FORMAT statement can be subscripted by a single optionally signed integer constant. All occurrences of such labels within a given block and having the same identifier constitute a declaration of a constant label array.

For example:

```
X(-2):
X(-1):
X(0):
X(2):
        .
        .
        .
        GOTO X(K);
```

In this example, X is a constant label array of bounds (-2:2). Element X(1) is not defined. The program is invalid if it attempts to refer to X(1).

## 4.3 The Declare Statement

### Recommended Forms

There are many valid forms of the DECLARE statement; however, there are three ways to use the DECLARE statement which are recommended because they improve program readability.

1. The recommended form of a simple DECLARE statement is

   DECLARE name attribute-list *[,name attribute-list]...* ;

   where name is an identifier and attribute-list specifies the characteristics of the name. The attributes in the attribute-list may be given in any order.

   For example:

   ```
   DECLARE A FIXED DECIMAL(3) AUTOMATIC;
   DECLARE B EXTERNAL STATIC FIXED BINARY(15),
             C CHARACTER(10);
   ```

2. Several names all having the same attributes may be "factored" to eliminate repeated specification of the same attribute for many names. The recommended form of a DECLARE statement with factored attributes is:

   DECLARE (name-1 name-2 ... name-n) attribute-list;

   where name-1 to name-n is a parenthesized list of declared names separated by commas followed by the attribute-list which specifies the characteristics of all of the names.

   For example:

   ```
   DECLARE (A,B,C) POINTER;
   ```

   is equivalent to

   ```
   DECLARE A POINTER, B POINTER, C POINTER;
   ```

3. The recommended form of structure declarations:

   The outermost structure is the major structure, and nested structures are substructures. The immediate components of a structure are members of the structure.

   A structure is specified by declaring the major structure name at level 1 followed by the names of the structure members with levels greater than 1. Level numbers may have leading zeros, but such zeros have no significance. Level numbers must be separated from the declared name by at least one blank or comment.

   A substructure at level n contains all following items declared with level numbers greter than n up to but not including the next item declared with a level number less than or equal to n.

For example:

```
DECLARE 1 INVENTORY EXTERNAL STATIC,
          2 PARTNO FIXED DECIMAL(6),
          2 DESCRIPTION,
                3 FINISH CHARACTER(10),
                3 SIZE,
                      4 LENGTH FIXED DECIMAL(3),
                      4 WIDTH FIXED DECIMAL(3),
                      4 HEIGHT FIXED DECIMAL(3),
                3 WEIGHT FIXED DECIMAL(10),
          2 QTY-ON-HAND FIXED DECIMAL(10),
          2 QTY-ON-ORDER FIXED DECIMAL(10);
```

DESCRIPTION is a substructure whose members are FINISH, SIZE.LENGTH, SIZE.WIDTH, SIZE.HEIGHT, and WEIGHT.

The storage class and scope attributes can only be given for the major structure. All components of the major structure have the same storage class and occupy a part of the same generation of storage. The scope of all member names is internal.

### Declaration of Arrays

If the attribute-list of a declared name contains the DIMENSION attribute the name is declared as an array whose dimensions and bounds are given by the DIMENSION attribute. The keyword DIMENSION is optional only if the DIMENSION attribute immediately follows the declared name. See the section which discusses the DIMENSION attribute in Section 4.4.

For example:

```
DECLARE A FIXED DECIMAL(3) DIMENSION(10);
```

or

```
DECLARE A(10) FIXED DECIMAL(3);
```

The DIMENSION attribute without a keyword can be factored only if it is the first attribute following a right parentheses.

For example:

```
DECLARE (X,Y)(10) FIXED;
```

becomes

```
DECLARE X(10) FIXED, Y(10) FIXED;
```

## Declaration of ENTRY Constants

Entry constants that identify PROCEDURE or ENTRY statements are declared by the label on the statement as described in Section 4.2. These names do not need to be declared in a DECLARE statement. However, if you wish to call an external procedure, you must use a DECLARE statement to create a declaration of the entry point name you wish to call.

For example:

```
EXT1:    PROCEDURE(A);
         DECLARE A FIXED;
         .
         .
         .
ENT1:    ENTRY(A);
         .
         .
         .
         END;

EXT2:    PROCEDURE RECURSIVE;
         DECLARE (EXT1, ENT1) ENTRY(FIXED);
         .
         .
         .
INT2:              PROCEDURE(A);
                   DECLARE A FIXED;
                   .
                   .
                   .
ENT2:              ENTRY(A);
                   .
                   .
                   .
                   END;
         .
         .
         .
         CALL EXT1(I);
         CALL ENT1(I);
         CALL EXT2;
         CALL INT2(I);
         CALL ENT2(I);
         .
         .
         .
         END;
```

EXT1 and EXT2 are external procedures. From within EXT2, calls are made to five procedures, so all five names must be declared. EXT2, INT2, and ENT2 are declared as entry constants by their use as labels on PROCEDURE or ENTRY statements. EXT1 and ENT1 are external entry points, so they must be declared as entry constants in a DECLARE statement. The declaration must describe each argument which is passed, and the value to be returned if the entry point is a function.

## 4.4 Alphabetic List of Attributes

This section contains descriptions of the attributes that can be used to declare the data type, storage class, and other characteristics of declared names. The use of some attributes implies others (see Section 4.5). Some attributes cannot be used in combination with others (Section 4.6). All names, other than major or minor structure names, must be declared with a data type attribute or BUILTIN.

Storage class attributes can be given only for level-one names (elementary data items, arrays, and major structures). For all level-one items that are not parameters for which no storage class is given, AUTOMATIC is assumed.

### ALIGNED

The ALIGNED attribute specifies that the associated data is allocated on a word boundary. When a generation of storage is to be shared or accessed by more than one name, all names used to access the generation must have the same alignment attribute. The ALIGNED attribute can only be specified with the CHARACTER or BIT attributes. Refer to Section 2.2 for more detail.

### AUTOMATIC

A name declared with the AUTOMATIC attribute is a variable whose storage class is AUTOMATIC. See Section 3.2, which discusses AUTOMATIC storage.

### BASED or BASED(r)

where r is a reference to a pointer variable or pointer valued function. A name declared with the BASED attribute is a variable whose storage class is BASED. See Section 3.4.

### BINARY or BINARY(p)

where p is an unsigned integer constant. A name declared with the BINARY attribute represents a binary arithmetic value of precision p. See Section 2.1 which discusses FIXED BINARY and FLOAT BINARY data.

### BIT or BIT(n)

where n, the length, is an *, an expression, or a REFER option.

The length specification indicates the length as follows:

- If * is given, the declared name must be a parameter. The actual length of the corresponding argument string is used as the length of the parameter.
- If the length is a REFER option, the declared name must be a member of a BASED structure. See Chapter 3 which discusses the REFER option.
- If the declared name is a STATIC variable, the length must be an integer constant.
- If the declared name is a parameter and the length is not *, then the length must be an integer constant.
- If the declared name is BASED, AUTOMATIC, or DEFINED, the length must be a fixed binary expression.

A name declared with the BIT attribute represents a bit-string value of length n. See Section 2.2 which discusses BIT and ALIGNED BIT data.

## BUILTIN

The BUILTIN attribute specifies that the name being declared is a BUILTIN function. It cannot be used with any other attribute. A name declared with the BUILTIN attribute must be one of the BUILTIN functions described in Chapter 10. It is not necessary to declare a BUILTIN function unless the name has been declared to mean something else in a containing block.

## CHARACTER or CHARACTER(n)

where n, the length, is an *, expression, or a REFER option.

The length specification indicates the length as follows:

- If * is given, the declared name must be a parameter. The actual length of the corresponding argument string is used as the length of the parameter.
- If the length is a REFER option, the declared name must be a member of a BASED structure. See Chapter 3 which discusses the REFER option.
- If the declared name is a STATIC variable, the length must be an integer constant.
- If the declared name is a parameter and the length is not *, then the length must be an integer constant.
- If the declared name is BASED, AUTOMATIC or DEFINED, the length must be a fixed binary expression.

A name declared with the CHARACTER attribute represents a character-string value of length n. See Section 2.2 which discusses CHARACTER, ALIGNED CHARACTER, and VARYING CHARACTER data.

## DECIMAL or DECIMAL(P) or DECIMAL(p,q)

where p and q are unsigned integer constants. A name declared with the DECIMAL attribute represents a decimal arithmetic value of precision (p,q). See Section 2.1 which discusses FIXED DECIMAL data.

## DEFINED(r)

A name declared with the DEFINED attribute is a variable whose generation of storage is identified by the reference r, which must not be bit-aligned. See Section 3.5 which discusses Storage Sharing by Defined Variables and Criteria for Storage Sharing.

## DIMENSION(b1,b2, ..., bn)

where n is less than or equal to 8; where each b is either lb:hb or hb or *; and where lb, the lower bound, and hb, the upper bound, are fixed binary expressions or REFER options.

The DIMENSION attribute specifies the number of dimensions of an array and the bounds of each dimension. The keyword DIMENSION is optional only if the DIMENSION attribute immediately follows the declared name.

The bounds specification indicates the bounds as follows:

- If only hb is given, the lb is 1.
- The lb must be less than or equal to the hb.
- If * is given, the declared name must be a parameter. In this case the actual bounds of the corresponding argument array are used as the bounds of the parameter array.

- If either lb or hb is a REFER option, the declared name must be a member of a BASED structure. See Section 3.4 which discusses the REFER option.
- If the declared name is a STATIC variable, lb and hb must be optionaly signed integer constants.
- If the declared name is a parameter and b is not *, lb and hb must be optionally signed integer constants.
- If the declared name is AUTOMATIC, BASED, or DEFINED, lb and hb must be FIXED BINARY expressions.

The DIMENSION attribute cannot be specified for file constants, entry constants, or built-in functions.

## ENTRY or ENTRY(att-1, att-2, ..., att-n)

where att-1 to att-n are the attribute-lists of parameters of an entry point.

A name declared with the ENTRY attribute represents entry values. See Section 2.5 which discusses ENTRY data.

If ENTRY is used with the VARIABLE attribute, the declaration is that of an ENTRY variable. The attribute lists must match the attribute lists of any ENTRY constant assigned to it.

Otherwise, the declaration is of an ENTRY constant that is an entry point in an external procedure other than the one containing the declaration. The attribute lists must describe the declarations, in the external procedure, of the parameters of the external entry point.

The EXTERNAL attribute is supplied to all declarations of ENTRY constants.

Each attribute list in the declaration describes the attributes of a single parameter; the parameter name is not listed. Factoring of attributes is not allowed. The attribute lists within an ENTRY attribute are also called *descriptors* or *argument descriptors.*

If the parameter is an array, the DIMENSION attribute must be the first attribute specified; the actual DIMENSION keyword is optional. Otherwise, attributes may appear in any order. If the parameter is a structure, the level number must precede the attribute-list for each level. Lengths and bounds may only be specified by decimal integer constants or asterisks.

For example:

If an external procedure begins:

```
P:  PROCEDURE(E,F,G);
    DECLARE 1 E,
                        2 X CHARACTER(20),
                        2 Y FIXED DECIMAL(3,2);
    DECLARE F(3,3) FIXED DECIMAL(3);
    DECLARE G POINTER;
```

then the declaration of P as an ENTRY constant in another external procedure would look like:

```
DECLARE A ENTRY(1,2 CHARACTER(20), 2 FIXED DECIMAL(3,2),
(3,3) FIXED DECIMAL(3), POINTER);
```

Following are more examples of parameter declarations and their corresponding attribute descriptors in an ENTRY constant declaration:

| Declaration of Parameter | Attribute Descriptor |
|---|---|
| DECLARE A FIXED BINARY; | ;FIXED BINARY, |
| DECLARE B(10) CHARACTER(5); | ,(10) CHARACTER(5), |
| DECLARE C(*) CHARACTER(*); | , DIMENSION(*) CHARACTER(*), |
| DECLARE 1 D(100),<br>2 E FIXED BINARY,<br>2 F(4,4)BIT(4); | ,1(100),2 FIXED BINARY,<br>2(4,4) BIT(4), |
| DECLARE G VARIABLE ENTRY (FLOAT<br>BINARY) RETURNS(FLOAT BINARY); | , VARIABLE ENTRY (FLOAT<br>BINARY) RETURNS (FLOAT BINARY), |

## EXTERNAL

The EXTERNAL attribute specifies that the scope of the declared name is EXTERNAL. See Section 4.1.

The EXTERNAL attribute can only be specified with the STATIC attribute; however, the compiler supplies it to all declarations of file constants and all external entry constants.

## FILE

A name declared with the FILE attribute represents a file value. See Section 2.6 which discusses FILE data.

## FIXED or FIXED(p) or FIXED(p,q)

where p and q are unsigned integer constants. A name declared with the FIXED attribute represents fixed-point arithmetic values of precision(p,q). See Section 2.1 which discusses FIXED BINARY and FIXED DECIMAL data.

## FLOAT or FLOAT(p)

where p is an unsigned integer constant. A name declared with the FLOAT attribute represents floating-point arithmetic values of precision (p). See Section 2.1 which discusses FLOAT BINARY data.

## INITIAL(i1,i2, ... ,in)

where each i is one of the following:

NULL, a character-string, a bit-string, or an arithmetic constant.

The INITIAL attribute specifies an initial value to be assigned to a STATIC variable prior to program execution.

Only one value may be specified for a scalar variable. The number of values specified in the INITIAL attribute of an array variable must be equal to the number of elements in the array. A structure variable can be initialized only by INITIAL attibutes specified for each of its structure members.

Values specified for an array are assigned to successive elements of the array in row-major order, i.e., the final subscript varies most rapidly (see Chapter 2).

The value NULL may be used only to initialize a pointer variable.

An arithmetic constant or NULL may be preceded by a parenthesized replication factor which specifies the number of times the constant is to be repeated in the initialization of the elements of the array. However, no such replication factor can be given for a character-string or bit-string.

For example:

```
DECLARE A(3) FIXED DECIMAL(3) STATIC INITIAL((3)0);
```

is equivalent to

```
DECLARE A(3) FIXED DECIMAL(3) STATIC INITIAL(0,0,0);
```

which sets A(1), A(2), and A(3) to 0.

The INITIAL attribute can only be specified with the STATIC attribute or on members of a STATIC structure.

### INTERNAL

The INTERNAL attribute specifies that the scope of the declared name is INTERNAL. See Section 4.1.

The INTERNAL attribute can only be specified with the STATIC attribute.

### LABEL

A name declared with the LABEL attribute is a LABEL variable and represents label values. LABEL constants cannot be declared in a DECLARE statement. See Section 2.4 which discusses LABEL data.

### PICTURE"p"

where p is a picture. A name declared with the PICTURE attribute represents arithmetic values stored as character strings described by the picture p. See Section 7.9 which discusses Picture Controlled Conversion and Section 2.1 which discusses PICTURE data.

### POINTER

A name declared with the POINTER attribute represents pointer values. See Section 2.3 which discusses POINTER data.

### RETURNS(d)

where d is one of the data types described in Chapter 2. The RETURNS attribute must be specified in a DECLARE statement for an entry name that is used in a function reference. It is used to describe the attributes of the function value returned when that entry name is invoked as a function.

The data type, d, must agree with the attributes specified in the PROCEDURE statement or ENTRY statement to which the entry name is prefixed.

The extent expressions that appear in d must be unsigned decimal integer constants.

The RETURNS attribute can only be specified with the ENTRY attribute.

### STATIC

A name declared with the STATIC attribute is a variable whose storage class is STATIC. See Section 3.3 which discusses STATIC storage.

### VARIABLE

The VARIABLE attribute indicates that the declared name is a FILE or ENTRY variable rather than a FILE or ENTRY constant.

The VARIABLE attribute can only be specified with the FILE or ENTRY attributes.

Example:

```
DECLARE F1 ENTRY (FIXED) RETURNS (BIT);
DECLARE EV ENTRY(FIXED) VARIABLE RETURNS (BIT);
DECLARE I FIXED;

A:     PROCEDURE(X) RETURNS(BIT);
       DECLARE X FIXED;
       DECLARE Y BIT;
       .
       .
       .
       RETURN(Y);
       END;

IF F1(I) THEN DO;  /* REFERENCE TO EXTERNAL PROCEDURE F1,
                   WHICH RETURNS A BIT VALUE */
EV=A;              /* ASSIGNS AN ENTRY CONSTANT TO AN
                   ENTRY VARIABLE */

IF EV(I) THEN DO;  /* REFERENCE TO PROCEDURE A */
.
.
.
EV=F1;             /* ASSIGNS AN ENTRY CONSTANT TO
                   AN ENTRY VARIABLE */

IF EV(I) THEN DO;  /* REFERENCE TO PROCEDURE F1 */
```

### VARYING

The VARYING attribute specifies that the declared name represents strings of varying lengths. The VARYING attribute can only be specified with the CHARACTER attribute. See Section 2.2 which discusses VARYING CHARACTER data.

## 4.5 Completion of Attribute-Lists

Unless a declaration was produced by a DECLARE statement that provides all attributes, the declaration has an incomplete attribute-list. Similarly, an argument descriptor or RETURNS attribute of an ENTRY constant or variable may have an incomplete attribute list. The attribute-list is completed according to the following rules:

- If DECIMAL or BINARY is specified without FIXED or FLOAT, FIXED is supplied.
- If FIXED or FLOAT is specified without BINARY or DECIMAL, BINARY is supplied.
- If a storage class is not specified and the declared name is not a parameter, AUTOMATIC is supplied.
- If the storage class STATIC is specified without the INTERNAL or EXTERNAL attribute, INTERNAL is supplied.
- If precision is not specified for FIXED BINARY, FIXED BINARY(15) is supplied.

- If precision is not specified for FIXED DECIMAL, FIXED DECIMAL(7,0) is supplied.
- If precision is not specified for FLOAT BINARY, FLOAT BINARY(21) is supplied.
- If CHARACTER or BIT is specified without length, 1 is supplied.

## 4.6 Attribute Consistency

Once the attribute-list is completed it is checked for consistency.

A declaration is inconsistent, and therefore invalid, if it contains more than one data type, more than one storage class, or violates any of the combinatorial constraints given in the discussion of attributes.

### Valid Data Types

FIXED BINARY
FIXED DECIMAL
FLOAT BINARY
CHARACTER
VARYING CHARACTER
ALIGNED CHARACTER
BIT
ALIGNED BIT
POINTER
LABEL
ENTRY[RETURNS]
ENTRY VARIABLE[RETURNS]
FILE
FILE VARIABLE
BUILTIN
structure

### Valid Storage Classes

AUTOMATIC
BASED
INTERNAL STATIC [INITIAL]
EXTERNAL STATIC [INITIAL]
DEFINED
parameter

## 4.7 The General Form of a DECLARE Statement

We urge you to use the three simple forms of the DECLARE statement given earlier in this chapter. However, for the sake of completeness we give the general form of a DECLARE statement and rules for transforming it to a simple DECLARE statement.

```
DECLARE d1,d2, ... ,dn;
```

where each d is

```
k name a1 a2 ... an
```

or

```
k (d1,d2, ... ,dn) a1 a2 ... an
```

where k is an optional level number, each a is an attribute, and name is a name to be declared.

For example:

```
DECLARE ((X FLOAT, Y FIXED)BINARY, Z POINTER)STATIC;

DECLARE 1 S STATIC, 2(X CHARACTER(3), Y BIT(5)) ALIGNED;
```

Factored DECLARE statements are transformed into simple DECLARE statements by copying the level number and attribute-list of the innermost set of parentheses onto all names contained within the parentheses, removing the parentheses, and continuing with the next innermost set of parentheses. The previous examples then become:

```
DECLARE X FLOAT BINARY STATIC,
             Y FIXED BINARY STATIC,
             Z POINTER STATIC;

DECLARE 1 S STATIC,
             2 X CHARACTER(3) ALIGNED,
             2 Y BIT(5) ALIGNED;
```

The DIMENSION attribute without a keyword can be factored only if it is the first attribute following a right parenthesis.

For example:

```
DECLARE (X,Y)(10) FIXED;
```

becomes

```
DECLARE X(10) FIXED,Y(10) FIXED;
```

A declaration is invalid if, after factoring, any data item is declared with duplicate attributes, conflicting attributes, or more than one level number.

## 4.8 Sample Declarations and Their Meaning

```
DECLARE X FILE;
```

X is an external file constant.

```
DECLARE Y FILE VARIABLE;
```

Y is a file variable whose storage class is AUTOMATIC.

```
DECLARE HYPOT ENTRY(FLOAT,FLOAT) RETURNS(FLOAT);
```

HYPOT is an EXTERNAL function name. The attributes of both parameters are FLOAT BINARY(21). The attributes of the value returned by the function are FLOAT BINARY(21).

```
DECLARE (A,B,C) FIXED DECIMAL(4,2);
```

A, B, and C are decimal arithmetic variables of precision (4,2).

```
DECLARE D FIXED BINARY(3);
```

D is a binary arithmetic variable of precision (3).

```
DECLARE L LABEL;
```

L is a label variable whose storage class is AUTOMATIC.

```
DECLARE E PICTURE "999V.99";
```

E is a pictured variable which represents a decimal value in the range 0.00 to 999.99.

```
DECLARE E ENTRY(FIXED) RETURNS(FLOAT);
```

E is an EXTERNAL procedure which will be called as a function with a FIXED BINARY(15) argument and return a FLOAT BINARY(21) value.

```
DECLARE E ENTRY VARIABLE;
```

E is an entry variable whose storage class is AUTOMATIC.

```
DECLARE ABS FIXED DECIMAL(3);
BEGIN;
DECLARE ABS BUILTIN;
```

In the outer block, ABS is a variable. In the inner block, ABS is a built-in function.

End of Chapter

# Chapter 5
# References

The value and storage of a variable, the value of a named constant, or the value returned by a function is represented in the text of an external procedure by a reference to a declared name. A declaration establishes the meaning of a name within a given region of the program known as the scope of the declaration. See Chapter 4, which discusses the scope of a declaration.

A reference is resolved by finding the declaration to which it refers. A reference is evaluated during program execution by locating the generation of storage or value represented by the declared name.

There are five types of references: simple references, subscripted references, structure qualified references, pointer qualified references, and function references.

The meaning of a reference depends on the form of the reference, the attributes of the declared name, and the context in which the reference occurs.

Normally, the evaluation of a reference that identifies a variable yields the value currently stored in the variable. However, in some contexts evaluation of a reference yields the address of a generation of storage or may simply refer to the declaration of a name.

## Contexts that Yield the Address of a Generation of Storage

Evaluation of a reference that identifies a variable yields the address of a generation of storage of the variable if the reference is:

- The target of an assignment statement.
- An index of a DO statement.
- An argument, passed by reference, of a CALL statement or function reference.
- An argument to the ADDR built-in function.
- The variable to be freed in a FREE statement.
- The variable in the pseudo-variables STRING, SUBSTR or UNSPEC.
- The reference in a SET option of an ALLOCATE statement.
- The reference in an INTO option of a READ statement.
- The reference in the FROM option of a WRITE or REWRITE statement.
- A variable in the I/O list of a GET statement.
- The reference in a REFER option evaluated as the target of the assignment of its evaluated extent. (This occurs during allocation of storage for the BASED structure that contains the REFER option. See Section 3.4.)

### Contexts that Yield a Declaration

Evaluation of a reference that identifies a variable yields an identification of the variable's declaration if the reference is the variable to be allocated in an ALLOCATE statement or in a SIZE built-in function.

### Contexts that Yield a Value

Otherwise, evaluation of a reference that identifies a variable yields the current value stored in a generation of storage of the variable.

It is an error to reference a variable whose genertion of storage has not been allocated. It is always an error to reference the value of a variable if no value has been assigned to the variable. The order of evaluation of the components of a reference is undefined and any program that depends on the order is in error.

## 5.1 Simple References

A *simple reference* is an identifier.

## 5.2 Subscripted References

A *subscripted reference* is an identifier that has been declared as the name of an array followed by a list of subscripts. The subscripts are separated by commas and enclosed in parentheses. A subscript is an expression whose evaluation must yield a FIXED BINARY integer. The number of subscripts must be equal to the number of dimensions of the array and the value of a subscript must fall within the bounds declared for that dimension of the array. See Chapter 2 which discusses array data and Chapter 4 which discusses the DIMENSION attribute.

For example:

```
A(3) FIELD(2,B,C) X(K+W(I))
```

## 5.3 Structure Qualified References

The name of a member of a structure can have a scope which overlaps another declaration of the same name. In order to avoid any ambiguity in referring to these similarly named items, it is necessary to create a unique reference called a qualified name. In a qualified name, the name common to more than one item is preceded by the name of the structure in which it is contained. This, in turn, is preceded by the name of its containing structure until the qualified name is a unique reference.

Thus, a qualified name is a sequence of names specified left to right in order of increasing level numbers. The names are separated by periods, and blanks may be placed around the periods. The sequence of names need not include all of the containing structures, but it must include sufficient names to avoid ambiguity. Any of the names may be subscripted.

If the sequence of names includes all of the structures containing the member with the rightmost name, then the name is fully qualified.

If the sequence of names includes only some of the names of the structures containing members with the rightmost name, then the name is partially qualified.

For example:

```
DECLARE 1 CENSUS,
          2 NAME,
                   3 LAST CHAR(20),
                   3 FIRST CHAR(20),
                   3 INITIAL CHAR(20);

DECLARE 1 DIRECTORY,
          2 NAME,
                   3 LAST CHAR(20),
                   3 FIRST CHAR(20),
                   3 INITIAL CHAR(20);
```

CENSUS.NAME.LAST and DIRECTORY.NAME.LAST are fully qualified references and CENSUS.LAST and DIRECTORY.LAST are partially qualified references. NAME.LAST is an ambiguous reference.

The elements of an array contained in a structure and requiring name qualification for identification are referred to by subscripted qualified names.

The subscripts used in a structure qualified reference do not have to appear immediately following the names to which they apply. As long as the order of the subscripts is preserved, the subscripts may be moved to the left or the right and written after any of the names in the reference. Use of this feature obscures the meaning of a program and should be avoided.

The number of subscripts must equal the number of dimensions of the referenced name including all dimensions inherited from structure members at higher levels.

For example:

```
DECLARE 1 S(3),
          2 G(4) FIXED,
          2 B FIXED;
```

A subscripted reference to G must contain two subscripts, (for example, S(1).G(3) or S.G(1,3)) and subscripted references to S or B must have one subscript (for example, S(3).B or S.B(3)).

## Reference Resolution

A fully or partially qualified reference is said to be applicable to declarations of structures that include the same hierarchy of structure names as is used in the qualified reference.

A declaration is applicable to a simple reference or subscripted reference if it is a declaration of the name given in the simple reference or subscripted reference.

References are resolved by looking for an applicable declaration in the block that immediately contains the reference. If no applicable declaration exists in that block, successive containing blocks are searched until a block containing an applicable declaration is found. The reference is in error if no applicable declaration exists in a containing block.

For example:

```
DECLARE 1 A,
          2 B,
                   3 C FIXED,
                   3 D FIXED;
BEGIN;
DECLARE 1 A,
          2 D,
                   3 E FIXED;
A.D = B.D;
```

A.D refers to D in the inner block. B.D refers to D in the outer block.

If a qualified reference is applicable to one and only one declaration in a block, that is the declaration to which the qualified reference refers. If the block contains more than one applicable declaration, the reference must be a fully qualified reference to one and only one of the declarations in the block.

For example:

```
DECLARE 1 A,
          2 B,
               3 C FIXED,
               3 E FIXED;
DECLARE 1 A,
          2 C FIXED,
          2 D,
               3 E FIXED;
```

A.C is a valid reference to the second A.C. A.E is an ambiguous reference.

If a nonmember name and a structure member name have the same identifier, the unqualified use of that identifier is understood to refer to the nonmember name. Reference to the structure member requires a structure qualified name.

For example:

```
DECLARE A FIXED;
DECLARE 1 S,
          2 A,
               3 B FIXED,
               3 C FIXED;
```

A refers to the FIXED variable, S.A refers to the substructure.

The presence or absence of subscript lists or argument lists does not affect the resolution of references and cannot resolve otherwise ambiguous references. Likewise, the data type or other attributes required by the context in which a reference appears do not affect resolution of a reference.

For example:

```
DECLARE X(5) FIXED;
DECLARE A FIXED;
BEGIN;
DECLARE 1 S,
          2 A FLOAT;
X(A) = 5;
```

The A in X(A) refers to S.A even though a fixed-point value is required by the context in which A appears.

A reference, Y, contained in a declaration of X, is resolved in the block that immediately contains the declaration of X, and is evaluated when X is referenced or allocated. Y is evaluated as if it were referenced in the block that immediately contains the reference to X or caused the allocation of X.

For example:

```
DECLARE X(10) FIXED DEFINED(Y);
DECLARE Y(10) FIXED;
BEGIN;
DECLARE Y POINTER;
X(K) = 5;
```

The reference X(K) refers to the DEFINED variable in the outer block. It in turn refers to the array Y in the outer block.

## 5.4 Pointer Qualified References

The format of a pointer qualified reference is:

```
pointer-expression -> based reference
```

The value of the pointer expression identifies a particular generation of storage whose attributes are described in this declaration of the based reference. A pointer expression is either a pointer-valued function or a reference to a pointer variable. A reference to a pointer variable can itself be pointer qualified as shown by the third example.

For example:

```
P -> X

DIRECTORY.ITEM(10,2) -> TABLE.ENTRY

P -> LINK.NEXT -> LINK.VALUE

ADDR(Y) -> Z(K)
```

It is an error to reference word-aligned data with a byte pointer or to reference byte-aligned data with a word pointer.

## 5.5 Function References

The format of a function reference is:

**entry-reference (arg1, arg2, ... , argn)**

or

**entry-reference()**

where **entry-reference** is either an entry constant or an entry variable, and **arg1** to **argn** is a parenthesized argument-list.

When a function reference appears in an expression, the procedure associated with the entry-reference is invoked using the arguments, if any, specified in the argument-list.

The result of this invocation is the value returned by the procedure.

For example:

```
C = F(R) + G(S);
```

Both F(R) and G(S) are function references. C is assigned the sum of the values returned by F and G.

A function reference may contain an entry-reference which is an entry variable. Since it is possible for an entry variable to be subscripted, a function reference may contain two parenthesized lists. The first list is the subscript of the entry-reference and the second is the argument-list of the function reference.

For example:

```
DECLARE E(10) VARIABLE ENTRY(FIXED,FIXED) RETURNS(FIXED);
.
.
A = E(3)(2,2) + B;
.
```

3 is the subscript and (2,2) is the argument-list.

Note that a function F which has no arguments must be invoked using an empty argument-list.

For example:

```
X = F();
```

A reference to F without an argument-list is a reference to the entry value, rather than a reference to the value returned by invoking F.

A procedure invoked by a function reference is a function and must return control either by the execution of a RETURN statement containing a return value or by the execution of a nonlocal GOTO statement.

For example:

```
Y:          PROCEDURE (X) RETURNS(FIXED BINARY);
            DECLARE X FIXED BINARY(15);
            DECLARE Z FIXED BINARY(15);
            .
            .
            .
            RETURN(Z);
            .
            .
            .
            END;
            .
            .
            .
            A = Y(5) + B;             /* invokes procedure Y as a
            .                         function and returns a FIXED BINARY
            .                         value*/
            .
```

If control returns from a function by the execution of a nonlocal GOTO statement, the execution of the statement which invoked the function is incomplete and is an error under the following conditions:

The evaluation of an extent expression of an AUTOMATIC or DEFINED variable can invoke a procedure. If control returns from the procedure via a nonlocal GOTO statement, the statement which invoked the procedure is incomplete, i.e., the evaluation of the extent expression is incomplete. It is an error for control to return to the block in which the AUTOMATIC or DEFINED variable is declared.

For example:

```
.
.
F:          PROCEDURE(I,L)RETURNS(FIXED);
            DECLARE I FIXED;
            DECLARE L LABEL;
            IF I < 0 THEN GOTO L;
            RETURN(100+I);
            END;
            .
            .
            .
            BEGIN;
DCL         X FIXED;
            .
            X = -25;
            .
            .
            BEGIN;
DCL         A(F(X,L2))FIXED;
            .
            .
L2:         .
            .
            .
            .
            END;
```

Evaluation of the extent expression of A is incomplete; therefore, it is an error for control to pass to L2. See Section 3.2 which discusses the evaluation of extent expressions for automatic variables.

## Built-In Function References

A built-in function is an intrinsic part of the PL/I language. All references to built-in function names are function references in that they refer to the value returned by the function. A built-in function name has no entry value and cannot be used in contexts that require entry values. Built-in functions that take no arguments may be referenced with or without an empty argument list. See Chapter 10 which discusses built-in functions.

## 5.6 Parameters and Arguments

An argument is an expression or data aggregate used in the argument-list of a CALL statement or function reference. A parameter is a name used in the parameter-list of a PROCEDURE statement. It is used by the invoked procedure to reference an argument. The Nth argument in an argument-list corresponds to the Nth parameter specified in the parameter list of the invoked entry without regard to the names of the parameters. The number of arguments and parameters must be the same. The correspondence between an argument and a parameter lasts until the block activation that established the correspondence is deactivated by a RETURN statement, nonlocal GOTO or END statement.

Items which can be passed as arguments include scalar expressions, arrays, and structures. An element of a structure array, a substructure, or an unconnected array may be passed. Cross-sections of multidimensional arrays cannot be passed as arguments.

All names used as parameters in a procedure must be declared as variables in that procedure.

Example:

```
P1:         PROCEDURE;
            DECLARE (A,B,C) FIXED;
            .
            .
            .
            CALL P2(A,B,C);
            .
            .
            .
            END;
P2:         PROCEDURE(A,X,B);
            DECLARE(A,X,B) FIXED;
            .
            .
            .
            END;
```

When P1 calls P2, argument A is associated with parameter A, argument B with parameter X, and argument C with parameter B.

### Argument Passing By Reference

When an argument is passed by reference the generation of storage of the argument is associated with the parameter. In other words, the same memory location is used by a called procedure for a parameter as is used by the calling procedure for the corresponding argument. Thus, changes to the parameter by the invoked procedure are reflected in the original argument.

An argument is passed by reference only when it is a reference to a variable whose attributes and extents match the attributes and extents declared for the parameter. The following attributes must match:

FIXED, FLOAT, BINARY, DECIMAL, BIT, CHARACTER, PICTURE, POINTER, LABEL, ENTRY, FILE, VARYING, and ALIGNED.

The precisions, lengths and pictures must also match, but other attribute information such as attribute-lists in an ENTRY attribute do not have to match.

If a parameter is an array, the argument must be an array with identical data type, bounds, and dimensionality.

If a parameter is a structure, the argument must be a structure of identical shape, size and member data types.

## Argument Passing By Value

Any expression that is not a reference to a variable is passed by value. If an argument is a reference to a variable whose attributes and extents do not match those of the parameter, it cannot be passed by reference and is passed by value. A literal constant, a reference to a named constant, and a reference enclosed in parentheses are considered expressions and are passed by value.

When an argument is passed by value, it is assigned to a temporary generation of storage in the calling procedure which is associated with the parameter. Changes to the parameter by the invoked procedure are not reflected back in the original argument. Similarly changes to the original argument do not affect the parameter.

However, because string constants are not copied into temporaries when they are passed by value, it is an error to assign a value to a parameter whose corresponding argument is a string constant.

When an argument is passed by value, it is converted to the data type of the parameter. If the argument cannot be converted to conform to the parameter, the program is in error. See Chapter 7 which discusses conversion of data types.

Arrays and structures cannot be converted and cannot be passed by value; they must be passed by reference.

## Asterisk and Constant Extents of Parameters

A parameter may be declared with either constant or asterisk extents. If a parameter is declared with asterisk extents, the asterisks are replaced by the extents of the argument that corresponds to the parameter. This replacement occurs each time the parameter is associated with an argument and holds only so long as the parameter remains associated with the argument.

For the purpose of determining whether an argument is to be passed by value or by reference, an asterisk extent is considered to match any extent of the argument.

An array parameter that corresponds to an unconnected array argument must be declared with asterisk extents. This means that if an unsubscripted member of a dimensioned structure is passed as an argument, the corresponding parameter must be declared with asterisk extents. See Section 2.7 which discusses unconnected arrays.

If a parameter is declared with constant extents, only arguments that have identical constant extents are considered to match the parameter.

## The Storage of a Parameter

Since the generation of storage associated with a parameter is always supplied by its corresponding argument, parameters are never allocated a generation of storage. The scope of a parameter is always internal to the block in which the name appears as a parameter.

It is an error to reference a parameter that is not associated with an argument.

For example:

```
A:        PROC(P1);
DECLARE   (P1,P2) FIXED;
.
.
.
B:        ENTRY(P2);
```

It is an error to reference P2 if this procedure was entered at A, or to reference P1 if this procedure was entered at B.

End of Chapter

# Chapter 6 Expressions

There are three kinds of expressions: primitive expressions, prefix expressions, and infix expressions. All expressions produce scalar values.

### Primitive Expressions

A *primitive expression* consists of a constant, variable, or function reference. Since primitive expressions contain no operators, their evaluation yields the value of the constant, variable, or function. For example: 5, Y.Z(K), F(X).

### Prefix Expressions

A *prefix expression* consists of a prefix operator (+, -, ^) followed by an expression called the *operand.* A prefix expression is evaluated by first evaluating the operand and then applying the operator to the value of the operand. For example: ^B, +5.1E3, -7.

### Infix Expressions

An *infix expression* consists of two expressions, called operands, separated by an infix operator. An infix expression is evaluated by first evaluating the operands and, if necessary, converting them to a common data type acceptable to the operator. Then the operator is applied to the converted value of the operands. For example: A + 2, X/Y.

## 6.1 The Order of Evaluation

The order of evaluation of an expression is determined by the priority of operations and by the use of parentheses.

In the evaluation of an expression, the priority of operations is as follows:

```
Highest: **  ^  prefix +  prefix -
         *  /
         infix +  infix -
         !!
         =  ^=  <  ^<  >  ^>  <=  >=
         &
Lowest:  !
```

Operations within an expression are performed in the order of decreasing priority. For example, in the expression, X + Y **2, exponentiation is peformed before addition. If an expression involves operations of the same priority, they are evaluated from left to right, except for prefix operators and the ** operator which are evaluated from right to left.

Subexpressions enclosed in parentheses are evaluated according to the rules for priority of operators and then treated as a single operand. Thus, parentheses may be used to modify the rules of priority. For example, in the expression A + B * C, multiplication is performed before addition but in the expression, (A + B) * C, addition is performed before multiplication.

If the result of an operation or expression can be determined without evaluation of one or more of its operands, the operands are not necessarily evaluated. A program that depends on the full evaluation of all operands is in error and the result of its execution is undefined. Similarly, a program that depends on an operand not being evaluated is in error.

For example:

```
IF X = 0 ! A = Y/X THEN ...
```

It is an error to assume that Y/X will not be evaluated if X is equal to 0.

```
IF A & FUNC(X) THEN
```

If A is false, the call to function FUNC will not necessarily be made.

## 6.2 Arithmetic Operators

The prefix arithmetic operator are:

| | |
|---|---|
| + | plus (positive) |
| - | minus (negative) |

The infix arithmetic operators are:

| | |
|---|---|
| + | add |
| - | subtract |
| * | multiply |
| / | divide |
| ** | exponentiate |

### Operand Conversion for Arithmetic Operators

Arithmetic operators require arithmetic operands. However, if both operands are arithmetic but differ in data type (except in some cases of exponentiation) they are converted to a common arithmetic type. If one operand is FIXED and the other is FLOAT, the common type is FLOAT. If one operand is FIXED BINARY and the other is FIXED DECIMAL, the common type is FIXED DECIMAL. (Note that this will produce a warning from the compiler because standard PL/I would produce a FIXED BINARY result in this case.)

The precision of two operands may differ without causing any conversion.

Pictured data is considered to be FIXED DECIMAL data for purposes of expression evaluation.

If two integer constants are used as operands of an arithmetic operator, they are considered to be FIXED DECIMAL, and produce a FIXED DECIMAL result.

### The Results of Arithmetic Operators

After the conversions specified above have taken place, the arithmetic operation is performed.

#### Prefix Operations

The prefix operators plus and minus yield a result having the data type and precision of the operand. The value of the result of a plus operator is the value of the operand. The value of the result of a minus operator is the value of the operand with its sign reversed.

#### Infix Operators

If the converted operands are FLOAT BINARY the result is FLOAT BINARY, while the precision of the result is the greater of the precisions of the two operands.

If the converted operands are fixed-point values, the result depends on the operator and the converted operands as described by the following.

If the operands are FIXED DECIMAL, let (p,q) be the precision of the first operand, and (r,s) be the precision of the second operand. If the operands are FIXED BINARY let (p) be the precision of the first operand and let (r) be the precision of the second operand.

If the operation is addition or subtraction, the result is the sum or difference of the converted operands. The precision of the result is:

FIXED DECIMAL: (min(16,max(p-q,r-s)+max(q,s)+1),max(q,s))

FIXED BINARY: (min(15,max(p,r)+1))

If the operation is multiplication, the value of the result is the product of the converted operands. The precision of the result is:

FIXED DECIMAL: (min(16,p+r+1),q+s)

FIXED BINARY: (min(15,p+r+1))

If the divide operator is used with two FIXED BINARY operands, it will produce an integral FIXED BINARY result. Standard PL/I would produce a SCALED BINARY result. Use the DIVIDE built-in function to avoid a warning message from the compiler.

If the operands are FIXED DECIMAL and the operation is division, the value of the result is the quotient of the first operand divided by the second operand. The precision of the result is (16,16-p+q-s). The precision and scale factor of the operands must be such that the result does not have a negative scale factor. If the quotient exceeds the precision of the result, the least significant digits of the quotient are truncated to form the result. Note that the result always has the maximum precision allowed and that as many fractional digits are preserved as is allowed by the implementation. Use of these values as operands of other fixed-point computations can easily lead to situations that produce values that exceed the limits of the implementation.

For example:

```
1/3 +25
```

The division yields a fixed decimal (16,15) result equal to 0.333 ... 3. The addition also yields a fixed decimal (16,15) result, which is insufficient to hold the value 25.333...3.

The DIVIDE built-in function can be used to control the precision of the result of fixed-point division.

If the operation is exponentiation, assume X is the first operand and Y is the second operand. The result is the value of X raised to the power of Y. If Y is a decimal integer constant and if X is FIXED BINARY and if ((p+1)*Y-1) < +15, the result is FIXED BINARY of precision ((p+1)*Y-1). If Y is a decimal integer constant and if X is FIXED DECIMAL and if ((p+1)*Y-1) < =16, the result is FIXED DECIMAL of precision ((p+1)*Y-1,q*Y). In all other cases, X and Y are converted to FLOAT BINARY and the result is a FLOAT BINARY value whose precision is the maximum of the precisions of the converted operands.

## Special Cases of Exponentiation

The result of the exponentiation operation is normally a machine-dependent approximation to X raised to the power Y, where X is the first operand and Y is the second operand. However, for special cases, X**Y is defined as follows:

If X=0 and Y > 0, the result is 0.

If X=0 and Y < =0, the error condition is signalled.

If X^=0 and Y=0, the result is 1.

If X < 0 and Y is not an integer, the error condition is signalled.

## 6.3 Bit-String Operators

The bit-string operators are:

| | |
|---|---|
| ^ | complement (not) |
| ! | inclusive or (logical or) |
| & | and (logical and) |

Bit-string operators require bit-string operands.

### The Results of Bit-String Operators

If the operation is complement, the value of the result is the complement of the value of the operand (each 1 becomes 0 and each 0 becomes 1).

If the operands of the logical operators are of different lengths, the shorter operand is extended on the right with zero bits until it is the length of the longer operand. The length of the result will be this extended length. Each bit of the result is developed by performing the indicated logical operation on the corresponding bits of the two operands. The following table defines the logical operations for a given bit.

| First Operand | Second Operand | Result of And | Result of Or |
|---|---|---|---|
| 1 | 1 | 1 | 1 |
| 1 | 0 | 0 | 1 |
| 0 | 1 | 0 | 1 |
| 0 | 0 | 0 | 0 |

For example: if A is "010001"B, then ^A is "101110"B. If B is "1111"B, then A&B is "010000"B and A!B is "111101"B.

## 6.4 The Concatenate Operator

The concatenate operator is !!. It is an infix operator that yields either a bit-string or character-string.

If both operands are bit-strings, the result is a bit-string; otherwise, both operands are converted to character-strings and the result is a character-string.

The value of the result is the converted value of the first operand concatenated with the converted value of the second operand.

The result is a string whose type is the type of the converted operands and whose length is the sum of the lengths of the converted operands.

## 6.5 The Relational Operators

The relational operators are:

| | |
|---|---|
| = | equal |
| ^= | not equal |
| < | less than |
| ^< | not less than |
| <= | less than or equal |
| > | greater than |
| ^> | not greater than |
| >= | greater than or equal |

## Types of Comparison

Comparison using the relational operators is defined for all data types. Both operands must be scalar values and, with the following exceptions, be of the same data type:

- Comparison can be made between bit strings without regard to alignment.
- Comparison can be made between character strings without regard to alignment or to the VARYING attribute.
- Comparison can be made between operands with different arithmetic data types. The operands will first be converted to the common arithmetic type, as described in Chapter 6, according to the rules for conversion in Chapter 7.

Character-string, bit-string, and arithmetic values may be compared using any relational operator. Label, entry, pointer, and file values can only be compared using the equal and not equal operators.

Arithmetic values and pictured values are compared algebraically.

Character-string values are compared by extending the shorter operand on the right with blank characters until it is the length of the longer operand. The two strings are then compared from left to right using the ASCII collating sequence. Note that in the ASCII collating sequence, the value of any uppercase letter is less than the value of any lowercase letter, and the value of any numerical character is less than the value of any letter.

Bit-string values are compared by extending the shorter operand to the length of the longer operand by padding the shorter on the right with zero bits. The two operands are then compared from left to right with 0 comparing less than 1.

Label values are equal only when they identify the same statement and the stack frame of the same block activation. Note that a label value which identifies a label on a null statement is not equal to a label value which identifies any other statement.

Entry values are equal only when they identify the same entry and the stack frame of the same block activation.

Pointer values are equal only when they identify the same generation of storage, or when they are both null.

File values are equal only if they identify the same file-control block.

## The Result of Relational Operators

Relational operators compare the values of their operands and yield a bit-string of length 1. The value of the result is "1"b if the relationship is true; otherwise, the value of the result is "0"b.

End of Chapter

# Chapter 7
# Type Conversion

*Conversion* is the changing of the representation of a value from one data type to another.

All conversions are described in terms of a source, a target, and a result. The *source* is the item to be converted. It has a data type and a value. The *target* is the data type to which the value is converted. The *result* is the converted value, having the data type of the target.

The target is derived from the context in which the conversion occurs. For example, the left side of an assignment statement supplies the target used to convert the right side's value. Sometimes, the target is not completely specified. For example, a partial target can result from the use of the conversion built-in functions BINARY, BIT, CHARACTER, DECIMAL, FIXED, and FLOAT. The target is made complete by the application of default rules associated with each type of conversion.

The following types of conversions are defined:

Arithmetic data type and precision
Arithmetic to bit-string
Arithmetic to character-string
Bit-string to arithmetic
Bit-string to character-string
Character-string to arithmetic
Character-string to bit-string
Format controlled
Picture controlled

No conversions are defined for pointer, label, entry, or file data.

The following sections describe each type of data conversion, and list contexts in which the conversion occurs. The lists are not meant to be exhaustive. Other contexts in which conversions occur are mentioned throughout the manual. The rules defined, however, apply to all conversions.

## 7.1 Arithmetic Type and Precision Conversion

Arithmetic values can be converted from any arithmetic data type to any other. Arithmetic conversions occur in the following contexts:

- An arithmetic expression is assigned to an arithmetic variable by means of an assignment statement or by being passed as an argument. In this case the expression will be converted to the data type and precision of the target variable.
- An arithmetic expression is returned as the value of an arithmetic-valued function. In this case, the expression will be converted to the data type and precision specified in the RETURNS attribute of the functions entry point.
- The FIXED, FLOAT, BINARY, or DECIMAL built-in function is used with an arithmetic first argument. In this case the value of the argument is converted to the data type indicated by the following table:

| Data Type of Argument | Built-In Function | | | |
|---|---|---|---|---|
| | FIXED | FLOAT | BINARY | DECIMAL |
| FIXED BINARY | FIXED BINARY | FLOAT BINARY | FIXED BINARY | FIXED DECIMAL |
| FLOAT BINARY | FIXED BINARY | FLOAT BINARY | FLOAT BINARY | *FIXED DECIMAL |
| FIXED DECIMAL | FIXED DECIMAL | *FLOAT BINARY | FIXED BINARY | FIXED DECIMAL |

*Standard PL/I would produce a FLOAT DECIMAL result. The compiler will give a warning if you use these cases.

If the precision of the result (and scale for the DECIMAL built-in function) is not provided as arguments, they will be determined according to the table below.

- An arithmetic infix operator is used on operands with different data types. In this case, the operands are converted to a common arithmetic type, as described in Chapter 6. The precision of the target is determined by the following table.

| | Data Type of Target | | | |
|---|---|---|---|---|
| | FIXED BINARY | FLOAT BINARY | FIXED DECIMAL OR PICTURE | |
| Data Type of Source | Precision of Target | Precision of Target | Precision of Target | Scale |
| FIXED BINARY | p[1] | p | ceil[2] (p/3.32)+1 | 0 |
| FLOAT BINARY | min(15,p) | p | min (ceil (p/3.32),16) | 0 |
| FIXED DECIMAL or PICTURE | min (ceil (p*3.32)+1,15) | min (ceil (p*3.32),53) | p | q |

1 where p is the precision of the source and q is the scale of the source

2 The function MIN, MAX, CEIL, and FLOOR are used in the text to indicate the precision, scale, and length of values that are results of expressions and conversions. These functions are used with the same meaning as the built-in functions with the same names in the PL/I language. See Chapter 10 for definitions of these functions.

The result of an arithmetic conversion is the value of the source converted to the data type of the target. In most cases, the result is the same value as the source, but if the data type of the target differs from the data type of the source, the result may be an approximation of the value of the source.

If the precision is insufficient to represent the value of the source, the value is truncated.

The program is invalid if the target is FIXED and has insufficient precision to represent the integral digits of the result value.

Example:

```
DECLARE (A,B)  FIXED BINARY(15);
DECLARE (D)    FIXED DECIMAL(7,2);
B = 5;
D = 16.21;
A = B + D;
```

In the last statement, the + operator has operands of different data types. According to the rules in Chapter 6 and the above table, B is converted to a FIXED DECIMAL(6,0) value. The addition produces a FIXED DECIMAL(9,2) value (21.21) which is converted to FIXED BINARY(15) upon assignment to A. The result is truncated to 21.

Note that the conversion built-in functions cannot be used to truncate the value of an expression. They may be used in a PUT LIST statement to control the number of spaces in the character-string value, but if the arithmetic value is greater than the precision allows, the program is in error.

For example:

```
DECLARE (A,B) FIXED BIN(15);
A = 5000;
B = FIXED (A,5);
PUT LIST(B);

OUTPUT: 5000
```

The expression FIXED(A,5) produces a FIXED BINARY(5) value. However, FIXED(A,5) does not actually cause any of the bits in a to be lost or set to zero. So when FIXED(A,5) is converted back to FIXED BINARY(15) for assignment to B, its original value is unchanged.

## 7.2 Arithmetic to Bit-String Conversion

Arithmetic to bit-string conversion occurs whenever an arithmetic expression is assigned or passed to a bit variable, returned as the value as a bit-valued function, or used as the first argument of the BIT built-in function.

The conversion is performed in the following steps:

- Take the absolute value of the source.
- Convert that value to FIXED BINARY(n), where n is determined by the following table:

| **Source** | **Value of n** |
|---|---|
| FIXED BINARY(p) | p |
| FLOAT BINARY(p) | min(15,p) |
| FIXED DECIMAL(p,q) | min(15, ceil((p-q)*3.32)) |

  The program is in error if n is insufficient to represent the integral digits of the source.

- Convert that value to a bit-string of length n whose value is the string of binary digits that represent the FIXED BINARY(n) value.
- If the target length is greater than n, extend the result on the right with zero bits. If the target length is less than n, truncate the rightmost excess bits of the result.

Examples:

| Source Type | Source Value | Intermediate Value | Target | Result |
|---|---|---|---|---|
| FIXED BINARY(15) | 5 | "000000000000101"B | BIT(6) | "000000"B |
| FIXED BINARY(6) | 5 | "000101"B | BIT(6) | "000101"B |
| FIXED BINARY(6) | -5 | "000101"B | BIT(6) | "000101"B |
| FIXED BINARY(4) | 5 | "0101"B | BIT(6) | "010100"B |
| FIXED DECIMAL(5,2) | 17.4 | "0000010001"B | BIT(6) | "000001"B |
| FIXED BINARY(15) | 5 | "000000000000101"B | BIT(16) | "0000000000001010" |

## 7.3 Arithmetic to Character-String Conversion

Arithmetic to character-string conversion occurs whenever an arithmetic expression is assigned or passed to a bit variable, returned as the value of a character-valued function, used as the first argument of the CHARACTER built-in function, or output to a STREAM file by a PUT statement.

The source is converted to a character string whose value is given by one of the following cases:

- If the source is FLOAT BINARY(b), first convert the b binary digits that contain the precision (fraction) of the value to a FIXED DECIMAL(p) number with precision p = ceil(b/3.32), and convert the exponent of the value to a FIXED DECIMAL(2) number. The value of the character-string result consists of p+6 characters: a minus sign if the source value is negative, otherwise a blank, followed by the most significant decimal digit of the fraction (0 if the value is zero), followed by a decimal point, followed by the remaining p-1 decimal digits of the fraction, followed by E, followed by the sign of the exponent, followed by the 2 digits of the exponent.

- If the source is FIXED BINARY(b), first convert the source to a FIXED DECIMAL(p) number, where p = ceil(b/3.32) + 1. The result is a character-string of length p+3 and its value consists of: the digits of the decimal representation of the value without leading zeros (the value zero has one zero digit), preceded by a minus sign if the source value is negative, preceded by sufficient blanks to form a string of p+3 characters.

- If the source is FIXED DECIMAL(p,q) and q=0, the length of the character string result is p+3 and its value consists of: the digits of the source value without leading zeros (the value zero has one zero digit), preceded by a minus sign if the source value is negative, preceded by sufficient blanks to form a string of p+3 characters.

- If the source is FIXED DECIMAL(p,q) and q > 0, the length of the character string result is p+3 and its value consists of: q fractional digits of the source value, preceded by a decimal point, preceded by the integral digits of the source value without leading zeros (fractional values and zero have one zero digit here), preceded by a minus sign if the source value is negative, preceded by sufficient blanks to form a string of p+3 characters.

If the target length is greater than the length of the character-string formed by the above rules, the character-string is extended on the right with blanks to form the target.

If the target length is less than the length of the character-string formed by the above rules, the rightmost excess characters are truncated.

Examples:

| Source Type | Source Value | Intermediate Value | Target | Result |
|---|---|---|---|---|
| FIXED BINARY(15) | 256 | "□□□□□□256" | CHAR(10) | "□□□□□□256□" |
| FIXED BINARY(15) | -3245 | "□□□□-3245" | CHAR(9) | "□□□□-3245" |
| FIXED BINARY(15) | -3245 | "□□□□-3245" | CHAR(5) | "□□□□-" |
| FIXED DECIMAL(5,3) | -3.22 | "□□-3.220" | CHAR(10) | "□□-3.220□□" |
| FLOAT BINARY(21) | 252.3E0 | "□2.523000E+02" | CHAR(15) | "□2.523000E+02□□" |
| FLOAT BINARY(8) | 252.3E0 | "□2.52E+02" | CHAR(9) | "□2.52E+02" |

## 7.4 Bit-String to Arithmetic Conversion

Bit-string to arithmetic conversion occurs whenever a bit-string expression is assigned or passed to an arithmetic variable, returned as the value of an arithmetic-valued function, or used as the first argument to the FIXED, FLOAT, BINARY, or DECIMAL built-in function.

The FIXED or BINARY built-in function will return a FIXED BINARY value. If the precision is not specified, it will be 15. The FLOAT built-in function will return a FLOAT BINARY value. If the precision is not specified, it will be 15. The DECIMAL built-in function will return a FIXED DECIMAL value. If the precision is not specified, it will be 6.

If the source is a null string, the value of the result is zero.

If the length of the string is n and 0 < n < =15, then consider the string to be an unsigned binary integer with precision n. The value of that integer is converted to conform to the data type and precision of the target.

The program is in error if the length of the bit-string is greater than 15.

Examples:

| Source | Target | Result |
|---|---|---|
| "101"B | FIXED BINARY(15) | 5 |
| "101"B | FIXED DECIMAL(10,3) | 5.000 |
| "101"B | FLOAT BINARY(21) | 5.0E0 |
| ""B | FIXED BINARY(15) | 0 |
| "00000"B | FIXED BINARY(15) | 0 |

## 7.5 Bit-String to Character-String Conversion

Bit-string to character-string conversion occurs whenever a bit-string expression is assigned or passed to a character variable, returned as the value of a character-valued function, used as the first argument to the CHARACTER built-in function, or used in the output list of a PUT statement.

If the source is a null bit-string, the intermediate result is a null character-string; otherwise, the source is converted to a character-string the same length as the source. If the Nth bit of the source is 0, the Nth character of the character-string is 0; and if the Nth bit of the source is 1, the Nth character of the character-string is 1.

If the target length is greater than the length of the character-string, the character-string is extended on the right with blanks until it is the length of the target.

If the target length is less than the length of the character-string the excess rightmost characters are truncated.

Examples:

| Source | Target | Result |
|---|---|---|
| "010110"B | CHARACTER(6) | "010110" |
| "010110"B | CHARACTER(3) | "010" |
| "010110"B | CHARACTER(9) | "010110□□□" |
| " "B | CHARACTER(3) | "□□□" |

## 7.6 Character-String to Arithmetic Conversion

Character-string to arithmetic conversion occurs whenever a character expression is assigned or passed to an arithmetic variable, returned as the value of an arithmetic-valued function, used as the first argument to the FIXED, FLOAT, BINARY, or DECIMAL built-in function, or when an arithmetic variable is input from a stream file with the GET statement.

The FIXED or DECIMAL built-in function will return a FIXED DECIMAL value. If the precision is not specified, it will be 16. The BINARY built-in function will return a FIXED BINARY value. If the precision is not specified, it will be 15. The FLOAT built-in function will return a FLOAT BINARY value. If the precision is not specified, it will be 53. (Standard PL/I would return a FLOAT DECIMAL value.)

If the source is a null string or contains all blanks, the value of the result is zero.

If the source is not a null string or all blank, it must contain a valid arithmetic constant (with an optional sign) in exactly the same form as such constant would appear in the text of a procedure: (Embedded blanks are not allowed.) Leading and trailing blanks are ignored. The source is converted to an arithmetic value that conforms to the data type and precision of the target.

The error condition will occur if the character-string is invalid, if the target data type is FIXED and its precision is insufficient to represent all of the integral digits of the converted value, or if the target data type is FLOAT and the value is too small or too large to be represented.

Examples:

| Source | Target | Result |
|---|---|---|
| "00123" | FIXED BINARY(15) | 123 |
| "00-45.63" | FIXED DECIMAL(8,3) | -45.630 |
| "-.12E-6" | FLOAT BINARY(21) | -.12E-6 |
| "-.12E-6" | FIXED DECIMAL(12,10) | -0.0000001200 |
| "-.12E-6" | FIXED BINARY(15) | 0 |
| "1.63" | FIXED BINARY(15) | 1 |
| "1.63E1" | FIXED BINARY(15) | 16 |
| "□1.63□□" | FIXED DECIMAL(7,0) | 1 |
| "2356.7" | FIXED DECIMAL(6,3) | (error) |
| "234X" | FIXED BINARY(15) | (error) |

## 7.7 Character-String to Bit-String Conversion

Character-string to bit string conversion occurs whenever a character-string expression is assigned or passed to a bit variable, returned as the value of a bit-valued function, or used as the first argument to the BIT built-in function.

If the source is a null character-string, it is first converted to a null bit-string; otherwise, the source is converted to a bit-string the same length as the source. If the Nth character of the source is 0 the Nth bit of the bit-string is 0 and if the Nth character of the source is 1, the Nth bit of the bit-string is 1. If the character string contains any character other than 0 or 1, the error condition is signalled.

If the target length is greater than the length of the bit-string, the bit-string is extended on the right with zeros.

If the target length is less than the length of the bit-string, the rightmost excess bits are truncated.

Examples:

| Source | Target | Result |
|---|---|---|
| "010110" | BIT(6) | "010110"B |
| "010110" | BIT(3) | "010"B |
| "010110" | BIT(9) | "010110000"B |
| " " | BIT(3) | "000"B |
| "01020" | BIT(3) | (error) |

## 7.8 Format Controlled Conversion

Format controlled conversion occurs when a GET or PUT statement containing a GET EDIT or PUT EDIT is executed. This section describes each of the format items that can appear in the format list of a GET EDIT or PUT EDIT statement, or in a FORMAT statement.

Input conversion occurs when a format specification is used to control conversion from a source field. Output conversion occurs when a format specified is used to control conversion to a result field. The result of an input conversion is assigned to a list element and, consequently, is converted again to conform to the data type of the list element. Refer to the sections on GET and PUT statements in Chapter 9.

### Fixed-Point Format

F(w) or F(w,d)

where w is an integer constant representing the length of the field.

d is an integer constant representing the decimal point location.

#### Fixed-Point Input Conversion

Let S be the source field to be converted to a FIXED DECIMAL result, X, according to the fixed format specified, F(w) or F(w,d).

The length of S is w. If w=0 or S contains all blanks, the result is zero; otherwise, S must be a character representation of an optionally signed, FIXED DECIMAL constant with optional leading or trailing blanks. (Embedded blanks are not allowed.) If S contains an invalid FIXED DECIMAL constant, the error condition is signalled.

S is converted to FIXED DECIMAL(p,q) where p is the number of digits in S. If S contains a ., q is the number of digits following the . ; otherwise, q is the value of d.

Examples:

| Value of S | Format | Value of X |
|---|---|---|
| □7.2□ | F(5) | 7.2 |
| □□□□□ | F(5) | 0 |
| -7□□□ | F(5) | -7 |
| □10.5 | F(5,2) | 10.5 |
| □100□ | F(5,2) | 1.00 |

### Fixed-Point Output Conversion

Let X be the value to be converted to a result field, S, according to the fixed format specified, F(w) or F(w,d).

If w=0, S is a null string.

If X is FIXED DECIMAL(16,d), the conversion to Y in the two cases below is omitted because no rounding can be performed. In this case, X is converted directly to Z.

If d=0, X is converted to a FIXED DECIMAL(16,1) value Y. Y+.5 is then converted to a FIXED DECIMAL(p) value Z, where p is min(16,w-1). Z is then converted to a character-string S that consists of: the digits of Z without leading zeros (the value zero has one zero digit), preceded by a minus sign if Z is negative, preceded by sufficient blanks to obtain a string of w characters.

For example:

| Value | Format | Result |
|---|---|---|
| 0 | F(4) | □□□0 |
| 52 | F(4) | □□52 |
| -7 | F(4) | □□-7 |
| 13.5 | F(4) | □□14 |
| -.5 | F(4) | □□-1 |
| 17.08 | F(4) | □□17 |

If d^=0, X is converted to a FIXED DECIMAL(16,d+1) value Y. Y is then rounded. The rounded value of Y is then converted to a FIXED DECIMAL(p,d) value Z, where p is min(16,w-2). Z is then converted to a character-string S that consists of the fractional digits of Z, preceded by a decimal point, preceded by the integral digits of Y without any leading zeros (if no itegral digits exist, a single zero digit is used), preceded by a minus sign if Z is negative, preceded by sufficient blanks to obtain a string of w characters. The program is invalid if the converted value of Z as described exceeds w characters in length.

For example:

| Value | Format | Result |
|---|---|---|
| 0 | F(6,1) | □□□0.0 |
| 52 | F(6,1) | □□52.0 |
| 4.05 | F(6,1) | □□□4.1 |
| -4.05 | F(6,1) | □□-4.1 |
| 8.005 | F(6,1) | □□□8.0 |
| -.005 | F(6,1) | □□□0.0 |
| .42 | F(6,2) | □□0.42 |
| -.75 | F(6,2) | □-0.75 |

## Floating-Point Format

E(w) or E(w,d)

where w is an integer constant representing the length of the field.

d is an integer constant representing the decimal point location.

### Floating-Point Input Conversion

Let S be the source field to be converted to a FLOAT BINARY result, X, according to the float format specified, E(w) or E(w,d).

The length of S is w. If w=0 or S is a string of all blanks, the result is zero; otherwise, S must be a character representation of an optionally signed arithmetic constant with optional leading and trailing blanks. (Embedded blanks are not allowed.) S is converted to a FLOAT BINARY value of precision 53.

Examples:

| Value of S | Format | Result |
|---|---|---|
| □□□□□ | E(5) | 0 |
| 1.4E7 | E(5,3) | .14E+8 |
| 12345 | E(5,3) | .12345E+2 |
| -2.5E-4 | E(7,0) | -.25E-3 |
| □35.E+10 | E(8) | .35E+12 |

### Floating-Point Output Format

Let X be the value to be converted to the result field S according to the float format specified, E(w) or E(w,d). X is converted to FLOAT BINARY(b) value Y according to the rules for type conversion given in Chapter 7. The b binary digits of Y are then converted to a FIXED DECIMAL(p) number Z, where p=ceil (b/3.32), and the exponent of Y is converted to a FIXED DECIMAL(2) number.

The length of S is w. If w=0, S is a null string.

If d is omitted, it is defined as p-1.

The result is a character-string S consisting of the two decimal exponent digits, preceded by the sign of the exponent, preceded by an E, preceded by d digits of Z (the second through d+1st digits), preceded by a decimal point if d^=0, preceded by the most significant digit of Z (the value zero is considered to have one signficant zero digit), preceded by a minus sign if Z is negative, preceded by sufficient blanks to form a string of w charcters. The program is invalid if the converted value of X as described above exceeds w characters in length.

For example:

| Value | Format | Result |
|---|---|---|
| 0 | E(10,3) | □0.000E+00 |
| -25 | E(10,3) | -2.500E+01 |
| 12345678 | E(10,3) | □1.234E+07 |
| 7.5E-10 | E(10,3) | □7.500E-10 |
| 0 | E(14) | □□0.000000E+00 |
| -25 | E(14) | □-2.500000E+01 |

In the last two examples, we assume that the decimal precision of the value to be converted is 7, thus giving a default value of 6 for d.

## Character-String Format

A or A(w)

where w is an integer constant representing the length of the field. For use in input formats, w must be given.

### Character-String Input Format

For input conversion, w must be specified. No conversion is perfomed. The result is a character string of length w.

Examples:

| Input Field | Format | Input Result |
|---|---|---|
| □□□□□ | A(5) | "□□□□□" |
| 2.5□ | A(4) | "2.5□" |
| 2XY□□2 | A(6) | "2XY□□2" |

### Character-String Output Format

For output conversion, w is optional. Let X be the value to be converted to the result field S. X is converted to a character string Y according to the rules for type conversion given in Chapter 7. If w is omitted, the result is Y; otherwise, Y is converted to a character-string of length w by truncating the excess rightmost characters or by padding on the right with blanks.

Examples:

| Value of X | Format | Value of S |
|---|---|---|
| "abc" | A | abc |
| "abc" | A(5) | abc□□ |
| "" | A(4) | □□□□ |
| "abc" | A(2) | ab |

## Bit String-Format

B(w) or B1(w) or B2(w) or B3(w) or B4(w)
B or B1 or B2 or B3 or B4

where w is an integer constant representing the length of the field. For use in input formats, w must be given.

### Bit String Input Format

Let S be the input field to be converted to the bit string X according to the bit string format specified.

For B(w) or B1(w) S must be a character string composed of the characters 0 and 1.

For B2(w) S must be a character string composed of the character 0,1,2, and 3.

For B3(w) S must be a character string composed of the characters 0,1,2,3,4,5,6, and 7.

For B4(w) S must be a character string composed of the characters 0,1,2,3,4,5,6,7,8,9,A,B,C,D,E, and F.

Each of the characters of S is converted to a series of n bits, where n is 1 for B or B1 format, 2 for B2 format, 3 for B3 format, and 4 for B4 format, according to the table in the section on BIT data in Chapter 2.

For input conversion, w must be specified. The source field is converted to a bit string according to the rules for character string to bit string conversion given in this chapter.

Examples:

| Input | Format | Result |
|---|---|---|
| 010 | B(3) | "010"B |
| 32 | B2(2) | "1110"B |
| 65 | B3(2) | "110101"B |
| 9D | B4(2) | "10011101"B |

### Bit String Output Format

Let X be the value to be converted to the result field S. X is first converted to a bit-string Y using the type conversion rules given in Sections 7.2 and 7.7. Y is then padded on the left with sufficient zero bits to make the length of Y a multiple of k, where k is 1, 2, 3, or 4 as determined by the format. Y is then converted to a character-string Z by taking groups of k bits from left to right and selecting characters from the table given in the section on BIT data in Chapter 2. If w is omitted, the length of the result is the length of Y divided by k; otherwise, the result is formed by right justifying Z in a field of w blanks. In this case w must be sufficient to contain all of Z.

Examples:

| Value of Y | Format | Value of S |
|---|---|---|
| "00"B | B | 00 |
| "1"b | B(4) | □□□1 |
| ""B | B(4) | □□□□ |
| "1101"B | B2(2) | 31 |
| "110101"B | B3(2) | 65 |
| "10011101"B | B4(2) | 9D |
| "10111"B | B2(4) | □113 |

## Picture Format

P"picture"

where picture must be a valid picture string as described in Section 2.1.

For input, no actual conversion occurs.

For output conversion, a value is converted to a character-string according to the rules for picture editing described in the next section.

## 7.9 Picture Controlled Conversion

There are two kinds of picture controlled conversion: encoding and editing. Encoding occurs when a pictured value is converted to an arithmetic value. Editing occurs when an arithmetic value is assigned to a picture variable or output through a P format item.

### Picture Encoding

Let the source be the pictured value to be encoded, and let (p,q) be the precision of the encoded result where (p,q) are determined according to the rules given in Section 2.1.

If the string of decimal digits within the source is a null string, the result is zero; otherwise, the absolute value of the decimal integer represented by the string of decimal digits becomes the value of the result after being scaled by dividing it by 10**q. If the source contains a -,CR, or DB picture character the sign of the result is minus. The target is FIXED DECIMAL(p,q).

Examples:

| Source | Picture | Result |
|---|---|---|
| 12,345 | ZZ,ZZZ | 12345 |
| □□□900 | ZZ,ZZZ | 900 |
| □□5.00 | ZZZV.ZZ | 5.00 |
| □□5.00 | ZZZ.ZZ | 500 |
| □-5.00 | SSSV.99 | -5.00 |
| □+5.00 | SSSV.99 | 5.00 |
| □-5.00 | ---V.99 | -5.00 |

### Picture Editing

In order to prepare the source for editing, it is first converted to a FIXED DECIMAL(p,q) intermediate target, where p and q are determined according to the rules given in Section 2.1.

If this precision is insufficient to retain all digits to the left of the decimal point, the program is in error. If fractional digits are lost, they are truncated.

The converted source is then converted to a character-string by replacing the appropriate picture characters with the decimal digits of the absolute value of the converted source according to the rules for replacement given in Section 2.1.

If the value to be edited is zero and the picture does not contain at least one 9, the result is either all blank or all * depending on whether or not an * appears in the picture.

Negative numbers cannot be edited unless the picture contains a sign picture character.

Examples:

| Source | Picture | Result |
|---|---|---|
| 5.2 | ZZZVZZ | □□520 |
| 0.01 | ZZZVZZ | □□□01 |
| 0 | ZZZ | □□□ |
| 1234 | ZZZZV | 1234 |
| 12345 | 99999 | 12345 |
| 123 | 99999 | 00123 |
| -105.02 | $**,***V.99 | invalid |
| -105.02 | $**,***V.99CR | $***105.02CR |
| -75 | ----V-- | □-7500 |
| 75 | ----V-- | □□7500 |
| -20 | -999 | -020 |
| 20 | -999 | □020 |
| -275.03 | $$$$$V.99- | □$275.03- |
| 25.01 | $$$$$V.99- | □□$25.01□ |
| -7.5 | $$,$$$V.99DB | □□□□$7.50DB |
| 0 | -****V.** | ******** |
| 5 | -****V.** | □***5.00 |
| -75 | -****V.** | -**75.00 |
| .75 | Z.VZZ | □□75 |
| .75 | ZV.ZZ | □75 |
| 0 | ZZ$ | □□□ |

End of Chapter

# Chapter 8
# Input and Output

Input and output (I/O) statements enable data to be transmitted to and from a program. A collection of data external to a program is called a *data set.* A data set may be a disk file or an I/O device such as a printer or display terminal. The term *file* will be used interchangeably with data set.

There are two types of PL/I data sets: stream data sets and record data sets. Input and output operations are called *stream I/O* or *record I/O,* depending on which type of data set is being used.

## 8.1 Stream I/O

The data in a *stream data set* is a series of ASCII characters, organized into lines and pages. Lines are separated by a *linemark,* which is the ASCII new-line character. Pages are separated with a *pagemark,* which is the ASCII form feed. The series of characters is also called a *data stream.*

When a program does output to a stream file, the data is converted to CHARACTER, according to the rules for conversion in Chapter 7. When data is input from a stream file, it is converted from CHARACTER to the data type of the variable being read into. The data must contain a valid character-string for conversion to the variable's data type.

There are three ways to perform stream I/O: list-directed I/O, edit-directed I/O, and the READ/WRITE option with stream files.

### List-Directed I/O

*List-directed I/O* is done using the GET and PUT statements with the LIST option. GET LIST inputs a series of values to a list of variables specified in the statement. The values must be represented as character-strings in the file, but do not have to be in any particular format. They need only be separated by one or more spaces.

PUT LIST outputs a list of expressions to a stream file in a predefined format. Data output with PUT LIST may be input using GET LIST.

For examples and a complete description of GET LIST and PUT LIST, see the sections on the GET and PUT statements in Chapter 9.

### Edit-Directed I/O

*Edit-directed I/O* is similar to list-directed I/O. The difference is that the program specifies the exact format of the data to be input or output. Edit-directed I/O is done using the EDIT option of the GET and PUT statements.

PUT EDIT allows you to specify the format in which data is output. It can be used to arrange data in columns and print formatted page and column headings. It can specify fixed or floating-point format for arithmetic values, and the number of decimal places to be printed. GET EDIT allows you to input data which has been formatted. It can be used to read any section of an input line, and ignore the rest of the line.

For examples and a complete description of GET EDIT and PUT EDIT, see the sections of the GET and PUT statements in Chapter 9. The format specifiers used to format the data are described under the FORMAT statement in Chapter 9, and the rules governing conversion of data to the various formats are given in Section 7.8.

### READ and WRITE on STREAM Files

To facilitate the processing of variable length lines, Data General PL/I permits the READ and WRITE statements to be used with STREAM files. The READ statement will read a line of text into a scalar CHARACTER VARYING variable, and set the length word of the variable to the number of characters read, including the linemark or pagemark. The WRITE statement will write the current value of a CHARACTER VARYING variable to a stream file. The format of the statements are:

```
READ FILE(F) INTO(X);
WRITE FILE(G) FROM(X);
```

where F is a STREAM INPUT file, G is a STREAM OUTPUT file, and X is a scalar CHARACTER VARYING variable.

See the sections on the READ and WRITE statements in Chapter 9.

## 8.2 Record I/O

A *record data set* is a set of discrete pieces of data called records. Record I/O writes scalar or aggregate variables to a record file and reads records into variables without performing any conversion or formatting. Record output is used for data storage only; a record file cannot be printed. Record input is used to read the data written using record output. There are two types of record data sets and record I/O: sequential, and direct.

### Sequential I/O

A *sequential data set* is a sequence of records arranged in the order in which they were written. Each execution of a WRITE statement writes one record to the file. The records may vary in length. The READ statement is used to read a sequential file. Each execution of a READ statement reads one record. A record must be read into a scalar or aggregate variable of the same type and structure as the one from which the record was written.

For examples and details of performing sequential I/O, see the sections on the READ and WRITE statements in Chapter 9.

### Direct I/O

A *direct data set* is a set of records which can be written or read in any order. Each READ or WRITE statement must specify a record number, or *key,* which identifies the record to be read or written. Data is written from or read into a scalar or aggregate variable, as with sequential I/O. Each record may contain a different amount of data up to 512 bytes, but each record takes up 512 bytes in the file.

If a DIRECT file is opened with the UPDATE attribute (Chapter 8), the REWRITE and DELETE statements may be used, in addition to READ and WRITE. See the sections in Chapter 9 on the READ, WRITE, REWRITE, and DELETE statements.

## 8.3 File Control Blocks

A *file control block* is a block of memory which contains all the information necessary to define the relationship between the program and a data set. One file control block is allocated for every file constant declared in a program.

A FILE value consists of the address of, and therefore identifies, a file control block. A file constant identifies a specific file control block. A file variable or a file-valued function can identify any file control block.

When a file is opened (Section 8.4), a file control block is associated with, or attached to, a data set. The attributes of the file that are specified by the open statement are recorded in the file control block. These attributes will be referred to as if they were attributes of the file. The file control block also contains information that is updated on each I/O operation to the file.

A file control block contains the following information:

filename
title
currentrecord
filedescription
open/closed status
linesize
pagesize
columnposition
linenumber
pagenumber

These names are used throughout the manual to refer to the contents of the file control block.

The filename is the declared name of the file constant which identifies the file control block.

The title is the name of a file that is to be used as a PL/I data set. If a file is opened by an OPEN statement that contains a TITLE option, the title is the character string specified in the TITLE option. If a file is opened without the TITLE option, the title is the same as the filename; except if the filename is SYSIN or SYSPRINT, the title is the default system input or output filename.

For example:

```
DECLARE (F,G,K,L) FILE;
DECLARE H FILE VARIABLE;
DECLARE INFILE CHARACTER(10) VARYING;
OPEN FILE(F);                      /* OPENS A FILE NAMED F*/
OPEN FILE(G) TITLE("JUNE");        /* OPENS A FILE NAMED JUNE*/
INFILE="JULY";
OPEN FILE(L) TITLE(INFILE);        /* OPENS A FILE NAMED JULY*/
H = K;
OPEN FILE (H);                     /* OPENS A FILE NAMED K*/
```

The currentrecord is either null or it specifies a record in the data set identified by the title.

The filedescription is the set of attributes that describe the data set identified by the title. It is formed when the file control block is opened.

The open/closed status indicates whether the file is open or closed. Initially it is closed.

The linesize is the maximum number of data may be written between linemarks in a data stream attached to a file control block that has the STREAM and OUTPUT attributes. During stream output, a linemark is automatically inserted whenever a character is to be output and columnposition = linesize + 1. Linemarks are also output by evaluation of SKIP options, SKIP formats, LINE options, LINE formats, and COLUMN formats. See the sections in Chapter 9 which discuss the GET and PUT statements. If the LINESIZE option is specified, linesize is set to the value in the LINESIZE option. If LINESIZE is not specified for a STREAM OUTPUT file, linesize will be set to a default of 80.

The pagesize is the maximum number of linemarks that may be written without causing the ENDPAGE condition between pagemarks in a data stream attached to a file control block that has the PRINT attribute. During output on a data stream attached to a file that has the PRINT attribute, the ENDPAGE condition is signalled whenever linenumber = pagesize + 1. If the PAGESIZE option is specified, pagesize is set to the value in the PAGESIZE option. If PAGESIZE is not specified for a STREAM OUTPUT PRINT file, pagesize will be set to a default of 60.

The columnposition is the column into which the next character will be written, or the column from which the next character will be read. The input or output of a linemark sets the columnposition to 1.

The linenumber is a count of the number of linemarks output since the last pagemark, plus one. The output of a pagemark sets the linenumber to 1.

The pagenumber is a count of the pagemarks output since the file was opened, plus one. The initial value of pagenumber is 1. Pagenumber can be set by the PAGENO pseudo-variable. See Section 9.2.

## 8.4 Opening a File

Before I/O can be done to a file, the file must be opened.

### Using the OPEN Statement

A file may be opened explicitly by the use of the OPEN statement. The OPEN statement associates a data set with the file control block identified by the FILE reference of the statement. The file attributes specified in the OPEN statement are recorded in the file control block as the filedescription.

The use of certain file attributes in the OPEN statement implies others, according to the following list:

| **Attribute** | **Implied Attributes** |
|---|---|
| DIRECT | RECORD KEYED |
| KEYED | RECORD |
| PRINT | STREAM OUTPUT |
| SEQUENTIAL | RECORD |
| UPDATE | RECORD |

If, after the implied attributes are supplied, a required attribute is misssing from the filedescription, it is supplied by default, according to the following list:

| **Required Attributes** | **Default** | |
|---|---|---|
| STREAM or RECORD | STREAM | |
| INPUT or OUTPUT or UPDATE | INPUT | |
| SEQUENTIAL or DIRECT | SEQUENTIAL | (for RECORD files only) |

The implied and default attributes are added to the filedescription in the file control block.

If the filename is SYSPRINT and the filedescription contains the STREAM and the OUTPUT attributes, the PRINT attribute is added to the filedescription.

The filedescription must now consist of one of these valid sets of attributes:

```
STREAM INPUT
STREAM OUTPUT
STREAM OUTUT PRINT
RECORD INPUT SEQUENTIAL
RECORD OUTPUT SEQUENTIAL
RECORD INPUT DIRECT KEYED
RECORD OUTPUT DIRECT KEYED
RECORD UPDATE DIRECT KEYED
```

## Implicit File Opening

If a file is not open, it is opened implicitly by the use of the following statements:

| Statement | Opening Attributes |
|---|---|
| GET | STREAM INPUT |
| PUT | STREAM OUTPUT |
| READ | RECORD INPUT SEQUENTIAL |
| WRITE | RECORD OUTPUT SEQUENTIAL |

If a PUT statement implicitly opens filename SYSPRINT, the file will be opened with the STREAM OUTPUT PRINT attributes.

## Effects of Opening a File

When a file is opened, either explicitly or implicitly, the following actions take place:

- The filedescription in the file control block is set as described in the previous two sections.
- If the filedescription contains the OUTPUT attribute, any existing data set identified by the title is deleted and a new data set conforming to the filedescription is created.
- If the filedescription contains either the INPUT or UPDATE attributes, the data set identified by the title must exist, and must be compatible with the filedescription.
- If the filedescription contains the STREAM attribute, the columnposition is set to 1.
- If the filedescription contains the STREAM and OUTPUT attributes, the linesize is set to the value in the LINESIZE option if given, else it is set to 80.
- If the file contains the PRINT attribute, the linenumber and pagenumber are set to one. The pagesize is set to the value in the PAGESIZE option if given, else it is set to 60.
- If the filedescription contains the RECORD attribute and the INPUT or UPDATE attribute, the currentrecord is set to designate the first record in the data set. Otherwise it is set to null.
- The open/closed status is set to open.

## File Attributes

This table contains descriptions of the attributes which can be used in the OPEN statement, or which are supplied by implicitly opening a file.

| Keyword | Explanation |
|---|---|
| DIRECT | The records of the file must be accessed by the use of keys. |
| INPUT | The file can be used only for input. |
| KEYED | The records of the file must be accessed by the use of keys. |
| LINESIZE(l) | where l is a FIXED BINARY expression. The maximum number of characters per line of the file is l. May be specified for STREAM OUTPUT files only. The default linesize is 80. The maximum linesize is 132. |
| OUTPUT | The file can be used only for output. |

| | |
|---|---|
| PAGESIZE(p) | where p is a FIXED BINARY expression. The maximum number of lines per page without signalling ENDPAGE is p. May be specified for STREAM OUTPUT PRINT files only. The default pagesize is 60. |
| PRINT | The PRINT attribute will cause the ENDPAGE condition to be signalled when linenumber becomes equal to pagesize + 1. It also causes character strings to be output from a PUT LIST without the enclosing quote marks. If an attempt is later made to read the character-string with a GET LIST, any spaces will be treted as separators and each word of the string will be a separate data item. |
| RECORD | The data set consists of records. |
| SEQUENTIAL | Records are stored and read in the order in which they are written. |
| STREAM | The data set consists of a stream of ASCII characters. |
| TITLE (t) | where t is a CHARACTER expression. The value of t is the name of the file to be opened. The TITLE option must be used to open a data set whose name contains a character that is not a legal character for PL/I identifiers (e.g., @ , =, :). |
| UPDATE | The file can be used for both input and output. |

The following list summarizes the I/O statements and options which may be used on a file containing the various valid filedescriptions:

| **Filedescription** | **Valid I/O Operations** |
|---|---|
| STREAM INPUT: | GET<br>READ without KEY |
| STREAM OUTPUT: | PUT<br>WRITE without KEYFROM |
| STREAM OUTPUT PRINT: | PUT<br>WRITE without KEYFROM |
| RECORD INPUT SEQUENTIAL | READ without KEY |
| RECORD OUTPUT SEQUENTIAL | WRITE without KEYFROM |
| RECORD INPUT DIRECT KEYED | READ with KEY |
| RECORD OUTPUT DIRECT KEYED | WRITE with KEYFROM |
| RECORD UPDATE DIRECT KEYED | READ with KEY<br>WRITE with KEYFROM<br>REWRITE with KEY<br>DELETE with KEY |

## 8.5 Closing a File

A file is closed by the execution of a CLOSE statement. The open/closed status of the file control block identified by the FILE reference is set to closed, and the file control block is no longer associated with a data set.

The file control block may be reused with another data set. The data set may be reopened using a different filename and file control block.

The execution of a STOP statement or END statement that terminates the program closes all files.

End of Chapter

# Chapter 9
# Statements

This section describes each statement by showing the general form or syntax of the statement, and then describes the effect of evaluating the statement. If any part of a statement is optional, it is shown enclosed in square brackets "[]". The statements are described in alphabetical order.

## 9.1 The ALLOCATE Statement

```
ALLOCATE identifier SET(pointer-variable);
```

Execution of an ALLOCATE statement causes storage to be allocated for a based variable, and assigns the address of the newly allocated based variable to the pointer-variable referenced in the SET option.

The identifier must be the name of a level-one based variable (elementary variable, array, or major structure) that is to be allocated. The reference in the SET option must refer to a POINTER variable. See Section 3.4 which discusses BASED storage.

When the ALLOCATE statement is executed, the extent expressions in the DECLARE statement for the BASED variable are evaluated, and the required amount of storage is allocated. The location of this generation of storage is assigned to the pointer-variable following the SET keyword. Note that the amount of storage allocated depends on the current value of the extent expressions. If the values of the extent expressions change between allocations, the amount of storage allocated will also change.

For example:

```
DECLARE (P,Q) POINTER;
DECLARE L FIXED;
DECLARE ABC CHARACTER(L) BASED (P);
L = 5;
ALLOCATE ABC SET(P);
L = 10;
ALLOCATE ABC SET(Q);
```

This creates two strings ABC. The first is 5 characters long and referenced by P- > ABC (or just ABC). The second is 10 characters long and referenced by Q- > ABC.

## 9.2 The Assignment Statement

```
target = expression;
```

where **target** is either a reference or pseudo-variable.

Execution of an assignment statement evaluates an expression and assigns its value to a target variable.

Once the expression is evaluated, if the data type of the result does not match the data type of the target, the result will be converted to the data type of the target according to the rules for type conversion given in Chapter 7. If the result of the expression is a character-string and the target is arithmetic, the error condition will be signalled if the character-string contains an invalid value for conversion to arithmetic. If the result is a character-string and the target is BIT, the error condition will be signalled if the character-string contains any characters other than "0" or "1".

If the expression is a file constant, file variable, or file-valued function reference, the target must be a file variable. If the expression is a label constant, label variable, or label-valued function, the target must be a label variable. If the expression is a pointer variable or pointer-valued function, the target must be a pointer variable. If the expression is an entry constant, entry variable, or entry-valued function, the target must be an entry variable whose argument-descriptors and RETURNS-descriptor (if any) match in both number and attributes those of the entry value being assigned to it.

If the target is a BIT variable, the converted value is truncated on the right if it is too long or padded on the right with zero bits if it is too short. The resulting value is assigned to the target. If the target variable is a nonvarying CHARACTER string, the converted value is truncated on the right if it is too long or padded on the right with blanks if the value is too short. The resulting value is assigned to the target.

If the target is a VARYING string and the converted value is longer than the maximum length declared for the variable, the converted value is truncated on the right.

If the target is a VARYING string and the value of the expression is not greater than the maximum length, the value is assigned to the target.

If the expression is a reference to a scalar character-string or bit-string variable, or SUBSTR of a string variable, the generation of storage identified by that reference cannot overlap the generation of storage identified by the target, unless the target and the reference identify exactly the same generation of storage. See Section 3.5 which discusses storage sharing.

Evaluation of the expression cannot allocate, free, or assign a value to the target, or any generation of storage identified by any reference contained within the target, e.g., a subscript.

A reference to a structure or array can appear as an expression in the right side of an assignment whose target is a reference to a structure or connected array of identical shape, size, and component data types; otherwise, the expression must yield a scalar value and the target must be a scalar variable or a pseudo-variable.

## Pseudo-Variables

PAGENO(file-reference), STRING(reference), SUBSTR(reference, expression-i, expression-j) and UNSPEC(reference) are called pseudo-variables when they appear on the left-hand side of an assignment statement.

The reference in a PAGENO pseudo-variable must identify an open file with the PRINT attribute. The value of the expression is converted to FIXED BINARY(15) and assigned to the pagenumber value of the file-control block.

The reference in a STRING pseudo-variable must refer to either an array or a structure variable that consists entirely of CHARACTER data or that consists entirely of BIT data. The variable must not have the ALIGNED or VARYING attribute. If the variable is an array, it must be connected. The value of the expression is converted to a character-string or bit-string and assigned to the string variable identified by the reference.

Evaluation of the reference in a SUBSTR pseudo-variable must identify either a bit-string variable or a character-string variable. Evaluation of expression-i and expression-j of a SUBSTR pseudo-variable must yield FIXED BINARY values. If expression-j is omitted, its value is L-i+1, where L is the length of the string variable. The value of the expression on the right-hand side of the assignment statement is assigned to the string variable beginning with the i-th character or bit and continuing for a total of j characters or bits. All other characters or bits of the string are unmodified. The program is in error if i < 1 or i+j-1 is greater than the length of the string. The expression may not be a reference to the same string referenced by the SUBSTR pseudo-variable. It may be a reference to another SUBSTR of the string, if the two references do not overlap.

For example:

```
SUBSTR(A,2,3) = SUBSTR(A,10,3);
```

The reference in an UNSPEC pseudo-variable must refer to a generation of storage of a scalar or aggregate variable whose storage is connected. The generation of storage identified by the reference is treated as a bit-string variable, and the value of the expression, converted to a bit string if necessary, is assigned to it.

Examples:

```
A = B+C;
P->R1.ITEM = X(3)+ITEM_CODE;
PAGENO(DAILY_FILE) = PAGENO(DAILY_FILE)+1;
STRING(ARR) = LIST1 !! LIST2;
SUBSTR(ITEM_NAME,5,2) = ".X";
UNSPEC(A) = "4E40"B4;
```

## 9.3 The BEGIN Statement

```
BEGIN;
```

The BEGIN statement defines the start of a begin block. If executed, the BEGIN statement causes a block activation of the begin block. See Chapter 1 which discusses begin blocks and containment.

Because a label prefix on a BEGIN statement produces a declaration of a label constant rather than a declaration of an entry constant, a begin block cannot be invoked by the execution of a CALL statement or evaluation of a function reference.

The block activation of a BEGIN block is terminated when the END statement which ends the BEGIN block is executed, when a nonlocal GOTO statement transfers control outside of the BEGIN block, or when a block activation which is a predecessor of the BEGIN block is terminated.

When control returns from a begin block by execution of the END statement, it returns to the block immediately containing the begin block. Execution continues with the statement following the END statement.

If a RETURN statement within a begin block is executed, it returns control to the predecessor of the most recent procedure block activation. Thus, the current block activation is terminated and control returns either to the statement immediately following the invoking CALL statement or to the continued evaluation of the statement containing the invoking function reference. Section 9.22 discusses the RETURN statement.

For example:

```
X:      PROCEDURE;
        .
        .
        .
        BEGIN;
        .
        .
        .
                BEGIN;
                .
                .
                .
                RETURN;         /* RETURNS CONTROL TO A*/
                .
                .
                .
                END;
        END;
        END; /* X */
        .
        .
        .
        CALL X;
        .
A:      .
        .
        .
```

In this example, execution of the RETURN statement returns control to the statement labeled A.

A begin block can be used as an ON-unit. See Section 9.17.

## 9.4 The CALL Statement

```
CALL entry-reference;
```

Execution of the CALL statement transfers control to an entry point of a procedure and causes a block activation of that procedure.

Evaluation of the reference must yield an entry value that identifies an ENTRY or PROCEDURE statement that does not have the RETURNS attribute. If the entry reference requires an argument-list, the number of arguments must equal the number of parameters. Note that the arguments are evaluated in an undefined order and that programs which depend on the order of evaluation of the arguments are in error. If the entry reference does not require arguments, then it may or may not have an empty argument list. See Section 5.6 which discusses parameters and argument passing.

For example:

```
CALL F(X,Y,Z);
CALL G();
CALL G;
```

## 9.5 The CLOSE Statement

```
CLOSE FILE(file-reference);
```

Execution of the CLOSE statement closes the file identified by the reference.

Evaluation of the reference must yield a FILE value. Closing an unopened file has no effect.

For example:

```
CLOSE FILE(X);
CLOSE FILE(PO>Y(K));                 /* NOTE - Y is a BASED array of*/
                                     /* FILE VARIABLES */
```

## 9.6 The DECLARE Statement

A DECLARE statement specifies the attributes to be associated with a declared name. DECLARE statements cannot be used as clauses of THEN or ELSE statements nor can they be cases in a DO CASE statement. See Chapter 4 which discusses the DECLARE statement.

## 9.7 The DELETE Statement

```
DELETE (file-reference) KEY (expression);
```

Execution of the DELETE statement replaces a record in an UPDATE file with a record of all zero bits.

Evaluation of the file-reference must yield a file value. The file must be open and have the RECORD, UPDATE, and KEYED attributes.

The expression in the KEY option is converted to a CHARACTER(32) value. This value must be convertible to a FIXED BINARY value called a key. The currentrecord of the file-control block is set to identify the record designated by the key.

A record of all zero bits replaces the record identified by currentrecord.

For example:

```
DELETE FILE(X) KEY(Y);
```

## 9.8 The DO Statement

There are five types of DO statements; simple DO, DO WHILE, DO increment, DO list, and DO CASE.

The DO statement defines the start of a group. If the group is an iterative group (DO WHILE, DO increment, or DO list), execution of the DO statement causes repetitive execution of the statements within the group. A DO group may be used wherever a single statement is allowed.

All five types of groups are terminated by an END statement.

### Simple DO

```
DO;
```

This DO statement defines the start of a noniterative group. Execution of a simple DO causes execution of the statements within the group. See Section 1.2 which discusses groups.

## DO WHILE

DO WHILE(expression);

Execution of a DO WHILE loop causes the statements within the group to be executed repeatedly as long as the value of the expression is true. The expression is evaluated before each pass through the group. It must yield a scalar bit-string value. If any bit of the value is 1, the expression is true; otherwise, it is false.

For example:

```
I = 1;
DO WHILE(I <= 5);
.
.
.
I = I + 1;
END;
```

It is an error to GOTO a label inside of a DO WHILE group from outside the group.

## DO increment

DO reference = expression1 TO expression2 *[BY expression3];*

Execution of a DO increment causes the statements within the group to be executed with the variable, identified by the reference, set to the value of expression1 and increased by expression3 at the start of each subsequent pass through the group until the value of the reference exceeds the value of expression2. If the value of expression3 is negative, the value of the reference is decremented at the start of each subsequent pass through the group until the value of the reference is less than the value of expression2. If BY expression3 is omitted, +1 is assumed.

The evaluation of the expressions must yield FIXED BINARY values. The reference must refer to a FIXED BINARY variable.

The values of expression2 and expression3 are evaluated in an unspecified order after the value of expression1 is assigned to the reference. They are not re-evaluated each time the group is executed.

The test which compares the reference to expression2 is made before each pass through the group.

For example:

| | |
|---|---|
| DO I = 3 TO 2; | executed 0 times |
| DO I = 1 TO 5 BY 2; | executed 3 times |
| DO I = 1 TO 10; | executed 10 times |
| DO I = 10 TO 0 BY -3; | executed 4 times |

The value of the reference at the end of the execution of a DO group is the value it would have had on the next iteration. In the above examples, I would have values of 3, 7, 11, and -2, respectively, at the end of the execution of the DO group.

It is an error to GOTO a label inside of a DO increment group from outside the group.

## DO list

DO reference = expression1 *[, expression2 ... , expressionK];*

Execution of a DO list causes the statements within the group to be executed first with the variable identified by the reference set to expression1, again with the variable set to expression2, etc.

ExpressionK is evaluated just prior to the kth trip around the do-group and is not re-evaluated.

For example:

```
DO I = 1,3,10,11,15;
```

It is an error to GOTO a label inside of a DO list group from outside the group.

## DO CASE

DO CASE(expression); case1 case2 ... caseN END; *[OTHER WISE clause]*

This DO statement specifies that the group is composed of single statements, DO groups, or begin-blocks which are called cases. The first case is case 1.

The evaluation of the expression must yield a FIXED BINARY value, k. If k is less than 1 or greater than the number of cases within the group, N, and no OTHERWISE immediately follows the END of the group, the error condition is signalled. If k is less than 1 or greater than N and an OTHERWISE immediately follows the END of the group, control is transferred to the OTHERWISE clause. Otherwise, the kth case is evaluated. If control is not transferred elewhere by the kth case, then upon completion of the kth case, control is transferred to the statement immediately following the DO CASE statement. A case cannot be a DECLARE, FORMAT, PROCEDURE, or ENTRY statement.

For example:

```
DO CASE(I);

A = 10;                /* CASE 1*/

DO;                    /* CASE 2 */
A = 10;
B = 20;
END;

DO CASE(J);            /* CASE 3 */
I = 5;
I = 7;
END;

;                      /* CASE 4*/

A = 15;                /* CASE 5*/
END;
OTHERWISE DO;
I = 0;
A = 0;
END;
```

A GOTO statement may transfer control to a label inside a DO case, or from one case to another within a DO case. At the end of the case in which the label is contained, control will transfer to the statement following the DO case.

## 9.9 The END Statement

END;

The END statement terminates the block or group headed by the most recent DO, BEGIN, or PROCEDURE statement.

When an END statement that denotes the end of a procedure is executed, the current block activation is terminated and the preceding block activation becomes the current block activation. Control is transferred to the statement immediately following the CALL statement which invoked the procedure. It is an error to execute the END statement which denotes the end of a procedure if the procedure was invoked as a function. See Chapter 1 which discusses the flow of control.

When an END statement that denotes the end of a begin block is executed, the current block activation is terminated and the preceding block activation becomes the current block activation. If the begin block is an ON unit and the condition signalled is ENDFILE or ENDPAGE, control returns to the point where the signal occurred. If the condition signalled is the ERROR condition, a STOP statement is executed. See sections in this chapter which discuss ON units.

The effect of executing an END statement that denotes the end of a group depends on the DO statement that heads the group.

For example:

```
ABC:  PROCEDURE;
      .
      .
      .
      BEGIN;
      .
      .
      .
      END;                /* END OF BEGIN BLOCK*/
      .
      .
      .
      DO I = 1 TO 10;
      .
      .
      .
      END;                /* END OF GROUP*/
      .
      .
      .
      END;                /* END OF PROCEDURE*/
```

## 9.10 The ENTRY Statement

entryname: ENTRY *[(parameter-list)][RETURNS(data-type)];*

where *parameter-list* is (identifier *[,identifier]...*)

The ENTRY statement specifies a secondary entry point to a procedure.

The entryname is a single unsubscripted label which results in the declaration of an entry constant. A GOTO statement cannot transfer control to an ENTRY statement.

The parameter-list is a parenthesized sequence of identifiers separated by commas. The identifiers specify the parameters of the entry point. Each identifier must identify a level-one variable which is declared in the same block as the entry statement occurs.

If control is to pass to an ENTRY statement by the execution of a CALL statement, the ENTRY statement cannot have a RETURNS attribute. If control is to pass to an ENTRY statement by the execution of a function reference, the ENTRY statement must have a RETURNS attribute. See Chapter 5 which discusses parameters and arguments.

When control is transferred to the entry by the evaluation of a function reference or the execution of a CALL statement, a new block activation of the procedure occurs and the arguments of the function reference or CALL statement are associated with the parameters in the parameter-list. When control reaches an ENTRY statement as a result of completing the execution of the previous statement, control passes to the statement immediately following the ENTRY statement.

An ENTRY statement cannot be immediately contained by a BEGIN block or an iterative DO group.

For example:

```
E:          ENTRY(A1) RETUNS(CHARACTER(20));
```

The invocation of E requires one argument which is associated with the parameter A1 and results in a character-string of length 20.

## 9.11 The FORMAT Statement

The FORMAT statement specifies a format-list used to control the conversion of values transmitted to or from a stream data set during the execution of a GET or PUT statement containing the EDIT option. Refer to later sections in this chapter for a description of the GET and PUT statements.

```
formatname:          FORMAT(format-list);
```

where format-list is

*[integer]* format-item *[,[integer]format-item, ...]*

where *integer* is an unsigned integer constant k in the range 0 < =k < =255.

If the integer is specified, the format-item or format-list is repeated the number of times indicated by the integer. If the integer is zero the format-item or format-list is ignored. If the integer is omitted, it is assumed to be 1.

A format-item is either another format-list, or is one of the following:

| | |
|---|---|
| A *[(integer)]*<br>B *[(integer)]*<br>B1 *[(integer)]*<br>B2 *[(integer)]*<br>B3 *[(integer)]*<br>B4 *[(integer)]*<br>E(integer *[,integer]*)<br>F(integer *[,integer]*)<br>P "picture" | Data Format Items |
| R(identifier) | Remote Format Item |
| COLUMN(integer)<br>LINE(integer)<br>PAGE<br>SKIP *[(integer)]*<br>TAB *[(integer)]*<br>X(integer) | Control Format Items |

Any integer, k, contained in format-items must be an unsigned integer constant in the range 0 < =k < =255, unless otherwise constrained by the description of the particular format-item in the following sections.

The formatname is a single unsubscripted label. A GOTO statement cannot transfer control to a FORMAT statement.

Each time control passes to a format-list all format-items between the last used format-item and the next data format are evaluated, then the next data format item is evaluated and used to control the conversion of the value being transmitted to or from the stream data set.

If control reaches the end of a format-list in a FORMAT statement, control returns to the remote format-item that transferred control to the FORMAT statement.

If control reaches the end of the format-list of a GET statement or PUT statement and one or more values remain to be transmitted to or from the statements I/O list, control passes to the beginning of the format-list.

If a data list in a GET or PUT EDIT statement is exhausted before the end of the format-list, no further format-items are evaluated, even if the next format-item is a control format-item.

If control reaches a FORMAT statement as a result of normal execution of the preceding statement, control is transferred to the statement following the FORMAT statement.

## The Data Formats

Let k be the first integer specified in the F or E formats, the integer specified in the A, B, B1, B2, B3, or B4 formats or the number of characters described in the P format.

If the file has the INPUT attribute:

- The next k characters of the data stream are converted according to the rules for format controlled input conversion given in Section 7.8. The converted value is then assigned to the current target of the GET EDIT.
- If the field of characters to be converted contains one or more linemarks, the linemarks are ignored, except that each linemark causes the columnposition to be set to one.

If the file has the OUTPUT attribute:

- The current output value is evaluated and converted according to the rules for format controlled output conversion given in Section 7.8. The converted value is then placed in the data stream.
- If the converted value does not fit onto the current line, it is output by placing as much as will fit on the current line and the rest on subsequent lines. Each line is filled with as many characters as allowed by the linesize value of the file-control block.

---

### The Remote Format

R(label)

where label is a label on a FORMAT statement.

If control reaches a remote format, control is transferred to the format-list contained in the FORMAT statement identified by the label. When each format-item in the remote FORMAT statement has been used, control returns to the format-item following the remote format.

Example:

```
R1:         FORMAT(A(10), A(10), R(R2), A(10));
R2:         FORMAT(5F(7));
```

## The COLUMN Format

COLUMN(integer)

Execution of a COLUMN format positions the data stream to a specified column position relative to the beginning of a line.

The integer, k, specified must be greater than zero.

If the file has the INPUT attribute:

- If k < columnposition, the data stream is advanced to the next linemark. It is then advanced k-1 characters and columnposition is set to k; if a linemark is encountered before the kth character, the stream is positioned to the character following this linemark and columnposition is set to one.
- If k > = columnposition, k-columnposition characters are ignored and the columnposition is set to k. If a linemark is encountered before the kth character, the stream is positioned to the character following the linemark and columnposition is set to one.

If the file has the OUTPUT attribute:

- If K < columnposition, a linemark and k-1 blanks are placed into the data stream and columnposition is set to k.
- If k > = columnposition and k > linesize, a linemark is placed into the data stream and columnposition is set to one.
- If k > = columnposition and k < = linesize, k-columnposition blanks are placed into the data stream and columnposition is set to k.

## The LINE Format

LINE(integer)

Execution of a LINE format positions the data stream to a specified line relative to the top of a page. It can only be used on a PUT to a STREAM OUTPUT PRINT file.

The integer, k, specified must be greater than zero.

If k = linenumber and columnposition = 1, nothing is placed in the data stream and no file-control values are changed.

If k > linenumber and k < = pagesize, k-linenumber linemarks are placed into the data stream. The linenumber is set to k and the columnposition is set to one.

If k > pagesize or k < = line number < = pagesize, the ENDPAGE condition is signalled. If linenumber > pagesize, a pagemark is written into the data stream.

The error condition is signalled if the LINE format is used on a file which does not contain the PRINT attribute.

## The PAGE Format

PAGE

Execution of a PAGE format positions the data stream to the top of a new page. It can only be used on a PUT statement to a STREAM OUTPUT PRINT file.

A PAGE format is evaluated by placing a pagemark into the data stream, setting the linenumber and columnposition to one, and adding one to the pagenumber.

The ERROR condition is signalled if the PAGE format is used on a file which does not contain the PRINT attribute.

### The SKIP Format

SKIP *[(integer)]*

Execution of a SKIP format positions the data stream to the beginning of a line.

If the file has the INPUT attribute:

The integer k must be greater than zero. If integer is omitted, assume k = 1.

The data stream is advanced until k linemarks have been encountered. The stream is positioned to the character following the kth linemark and the columnposition set to one. If the end of the data stream is encountered during the scan, the ENDFILE condition is signalled.

If the file has the OUTPUT attribute:

The integer k must be greater than or equal to 0. If the integer is omitted, then k = 1.

If k = 0, then a carriage return is placed into the data stream.

If the file does not have the PRINT attribute, or if linenumber > pagesize, or if linenumber + k < = pagesize, then k linemarks are placed into the data stream. Each linemark sets columnposition to one and adds one to linenumber.

If the file has the PRINT attribute and linenumber < pagesize < linenumber + k, then pagesize - linenumber + 1 linemarks are placed into the data stream, and the ENDPAGE condition is signalled.

### The TAB Format

TAB *[(integer)]*

Execution of a TAB format positions the data stream to a tab stop. It can only be used on a PUT to a STREAM OUTPUT PRINT file.

The integer, k, specified must be > 0. If the integer is omitted, k = 1 is assumed.

A TAB format is evaluated by inserting sufficient blanks into the output stream to set columnposition to the kth tab stop relative to the current value of columnposition. The tab stops are columns 1, 9, 17, 25, etc.

The effect of linesize on output is described in the Data Formats earlier in this section.

The ERROR condition is signalled if the TAB format is used on a file which does not have the PRINT attribute.

### The X Format

X(integer)

Execution of an X format advances the data stream k columns relative to its previous position.

The integer, k, specified must be > = 0.

If the file has the INPUT attribute:

The next k characters of the stream are ignored.

If the file has the OUTPUT attribute:

K blanks are placed into the stream.

The effect of linemarks in input or linesize on the output is described in the Data Formats earlier in this section.

## 9.12 The FREE Statement

```
FREE reference;
```

Execution of a FREE statement causes the storage allocated for a level-one based variable to be released.

The reference must be a reference to an allocated level-one based variable.

Freeing a generation of storage makes any values represented in it undefined. It is an error with unpredictable consequences to attempt to access a freed generation of storage.

For example:

```
FREE P->X;
FREE A;
```

## 9.13 The GET Statement

The GET statement may take one of the following forms:

GET *[FILE(file-reference)]* SKIP *[(expression)];*

GET *[FILE(file-reference)] [SKIP[(expression)]]* LIST (input-list);

GET *[FILE(file-reference)] [SKIP[(expression)]]* EDIT(input-list)(format-list);

where input-list is input-item *[,input-item]...* An input-item is either a reference or input-list DO increment.

Format-list is described in Section 9.11.

Execution of a GET statement transmits data from a stream data set.

If the FILE option is not specified, FILE(SYSIN) is supplied and is attached to a default operating system filename. The user must ensure that the default operating system file exists. If the FILE option is specified, evaluation of the reference must yield a FILE value.

If the file is closed, it is implicitly opened with the STREAM and INPUT attributes.

If the SKIP option is specified, the value of the expression must be a FIXED BINARY integer, k. If the expression is omitted, SKIP(1) is assumed. The data stream is advanced until k linemarks have been encountered and columnposition is set to one.

Each time control passes to a GET EDIT or GET LIST statement, each input-item of the input-list beginning with the leftmost input-item is evaluated. This causes the data stream to be scanned and a value assigned to the input-item. If no more input-items remain, control transfers to the statement immediately following the GET statement.

If the next input-item is a reference, it is evaluated as if it were the target of an assignment statement.

If the next input-item is a parenthesized input-list followed by a DO increment, the DO increment controls the repetitive evaluation of the input-list as if the parenthesized output-list was a DO group. Refer to this chapter for a description of the DO increment.

Example:

```
GET LIST (A,(B(I) I=1 TO 7 BY 2));
```

is equivalent to

```
GET LIST (A,B(1),B(3),B(5),B(7));

GET LIST((A(J),(B(I,J) DO I = 1 TO 5) DO J = 1 TO 5));
```

is equivalent to

```
DO J = 1 TO 5;
GET LIST (A(J));
DO I = 1 TO 5;
GET LIST (B(I,J));
END;
END;
```

## GET LIST

If the next input-item is a scalar, it is selected as the next target. If the next input-item is an aggregate, each of its scalar components is selected to be a target. The elements of arrays are selected in row-major order and the members of structures are selected in the order in which they are stored in memory.

Once the next input-item is identified, the data stream is scanned to find the next nonspace. If the end of the data stream is encountered, the ENDFILE condition is signalled.

The effect of the following rules is to permit a data stream to contain a sequence of fields separated by blanks or commas. Each field can be either a quoted character-string constant, a quoted bit-string constant, or a string of characters that does not begin with a quote and that does not contain a comma or a blank.

After a field has been scanned, its characters are assigned to the current input target using the normal rules for PL/I assignment statements.

Note that if the target is a charcter-string variable, any field is valid. However, if the target is not a character-string variable, the field must be a valid value for conversion to the target according to the rules for character-string conversion given in Chapter 7.

If the next nonspace is a comma:

- If the last operation on this file was the execution of a GET LIST and its scan was stopped by a space, the scan for the next nonspace is continued; otherwise, this target is ignored. The effect of this rule is to make no assignment to list items that correspond to ,, fields in the data stream.

  For example:

  ```
  GET LIST (A,B,C,D);
  ```

- If the input stream contains 5, 6, 7 then 5 is assigned to A, 6 to B, and 7 to D. C will be unchanged.

If the next nonspace is not a comma or a quote:

- The data stream is scanned until the next space or comma. All of the characters scanned except the space or comma that stopped the scan are assigned to the target. If the end of the data stream is encountered during the scan, the ENDFILE condition is signalled. A linemark does not terminate a data item; scanning will continue on the next line.

If the next nonspace is a quote:

- The data stream is scanned to find a quoted bit-string or character-string constant; any linemark characters are removed and this string is concatenated with any subsequent characters scanned up to the next space or comma. If the end of the data stream is encountered during the scan, the ENDFILE condition is signalled.

- If the resulting string is a character-string constant, the surrounding quotation marks are removed and any contained pair of adjacent quotes are replaced by a single quote, and the resulting character-string value is assigned to the target.
- If the resulting string is a bit-string constant, the enclosing quotation marks and the trailing "B" are removed and the enclosed characters are converted to a bit-string value which is assigned to the target.
- If the resulting string is not a valid character-string or bit-string constant, the ERROR condition will be signalled.

  Examples:

  ```
  "0123A C"
  "0110110"B3
  "0123X"B3
  "01101"ABC
  ""
  ```

- The first example is a valid character-string constant. The second is a valid bit-string constant. The next two examples are not valid character-string or bit-string constants. The last example is the null character-string.

NOTE: Each linemark encountered during scanning of the data stream sets columnposition to one.

In order to prevent end of file from occurring during the scanning of a field, files must end with a blank or comma. All files written by PUT LIST end with a blank.

Examples:

```
GET FILE(F) SKIP LIST(X,Y,Z);
```

Execution of this statement causes the data stream identified by F to be advanced past the next linemark, then three values are obtained and assigned to X, Y, and Z.

```
GET LIST(A,B,(C(I) DO I = 1 TO 10));
```

Execution of this statement causes 12 values to be obtained from SYSIN and assigned to A, B, and C(1) to C(10).

```
X:          PROCEDURE;

            DECLARE A(3) CHARACTER(1);
            GET LIST(A);                          /* 3 CHARACTERS ARE OBTAINED */
                                                  /* FROM SYSIN AND ASSIGNED TO */
                                                  /* A(1), A(2), AND A(3) */
            END;              /* END X     */
```

## GET EDIT

The next input-item is established in the same way as in a GET LIST statement.

Once the target has been determined, control passes to the format-list. Evaluation of the format-list causes the data stream to be scanned and a value assigned to the target. See the section in this chapter which discusses format-lists. Any of the format items described in that section may be used, including the R format.

Examples:

```
GET EDIT (A,B,C) (F(5), 2(X(3),F5.2));
```

input line: 345.1□□□□200□□□□3.456□

The input fields are underlined. The effect of this GET EDIT is to assign the values 345.1, 2.00, and 3.456 to the variables A, B, and C.

```
R1:          FORMAT(X(3),F(5.2),X(3),F(5.2));
             GET EDIT(A,B,C) (F(5),R(R1));
```

This has the same effect as the first example.

## 9.14 The GOTO Statement

GOTO reference; or GO TO reference;

Execution of a GOTO statement transfers control to a labeled statement either within the current block activation or within a preceding block activation. A local GOTO statement transfers control to a labeled statement within the current block activation. A nonlocal GOTO statement transfers control to a labeled statement in a preceding block activation. See Section 1.3 which discusses block activation.

Evaluation of the reference must yield a scalar label value whose stack frame pointer identifies the current block activation or a preceding block activation.

It is an error to attempt to transfer control to a label value in an inactive block. It is an error to transfer control to a label within an iterative group from outside the group.

For example:

```
GOTO Z;
GOTO B(6);
GO TO C;
```

## 9.15 The IF Statement

IF expression THEN clause *[ELSE clause]*;

where clause is either an unlabeled statement, an unlabeled do-group, or an unlabeled begin block. A clause cannot be a DECLARE, FORMAT, PROCEDURE, END, or ENTRY statement.

Execution of an IF statement evaluates the expression and uses that value to transfer the flow of control to the THEN clause or to the ELSE clause or to the statement immediately following the IF statement.

Evaluation of the expression must yield a scalar bit-string value. If any bit of the value is 1, the THEN clause is executed; otherwise, the ELSE clause is executed if it is present. Control then passes to the statement immediately following the IF statement, unless the IF statement has transferred control elsewhere.

For example:

```
IF X^=Y THEN A=B;
```

Note that an ELSE clause must always be paired with a preceding THEN clause.

For example:

```
IF X=Y
         THEN IF A=B
                  THEN RETURN;
                  ELSE GOTO C;
```

The ELSE clause in this example is paired with the THEN clause of the second IF statement.

```
IF X=Y
        THEN IF A=B
                THEN RETURN;
                ELSE;
        ELSE GOTO C;
```

The second ELSE clause in this example is paired with the THEN clause of the first IF statement.

## 9.16 The Null Statement

```
;
```

The null statement is an empty statement which causes no action. It may be used anywhere that a statement is permitted, but is most useful as a THEN or ELSE clause or as a case in a DO CASE statement.

For example:

```
IF A > B
    THEN IF C=D
            THEN X=Y + Z;
            ELSE;
    ELSE X=10;
```

## 9.17 The ON Statement

```
ON condition ON unit
```

where condition may be ENDFILE(file-reference), ENDPAGE(file-reference), or ERROR.

where ON unit is an unlabeled begin block which defines the action to be taken for the condition signalled.

The execution of an ON statement causes the ON unit to be established, but does not cause execution of the ON unit. An established ON unit is associated with the block activation that contains the ON statement. An established ON unit remains established until it is reverted either by the execution of a REVERT statement within the same block activation that established the ON unit, by the termination of that block activation, or by the establishment of another ON unit for the same condition within the same block activation.

### Conditions

A condition is a state of the executing program. ENDFILE, ENDPAGE, and ERROR identify specific conditions which can be detected during program execution.

Control enters an established ON unit when the condition identified by the condition name is signalled.

A condition is signalled by the execution of a SIGNAL statement or by detection of the condition during program execution. A signal causes a block activation of the most recent ON unit established for the condition. If a condition is signalled and no ON unit has been established for that condition, a default ON unit for the condition is invoked. The action of each default ON unit is described with the discussion of each condition below.

Control returns from an ON unit by passing through the END statement or by execution of a nonlocal GOTO statement as described in the section on begin blocks. It is an error to attempt to return from an ON unit by executing a RETURN statement.

It is an error for an ON unit or any procedure called by the ON unit to free or assign to a variable that is being used in the statement from which the signal occurred. It is also an error for an ON unit or any procedure called by the ON unit to close the file for which an I/O condition (ENDFILE or ENDPAGE) was signalled if the ON unit subsequently returns to the source of the signal.

### The ENDFILE Condition

ENDFILE(file-reference)

This condition is signalled when a GET or READ statement attempts to read beyond the end of a stream data set or a READ statement attempt to read beyond the end of a record data set.

Evaluation of the file-reference must yield a FILE value.

Just before the ENDFILE condition is signalled, the filename is assigned to the ONFILE built-in function.

The end-of-file status remains until the file is closed. Repeated GET or READ statements will immediately signal the ENDFILE condition.

On return from the ON unit, control returns to the statement immediately following the GET or READ statement.

The default ON unit writes an error message to the user console and signals the ERROR condition.

### The ENDPAGE Condition

ENDPAGE(file-reference)

This condition is signalled when a PUT statement places a linemark into the data stream of a PRINT file and the newly updated linenumber equals the pagesize+1. (The last line of the page has been written.) Linenumber and pagesize are values of the file-control block identified by the value of the reference.

Evaluation of the file-reference must yield a FILE value.

Just before the ENDPAGE condition is signalled, the filename is assigned to the ONFILE built-in function.

When ENDPAGE is signalled, the linenumber is one greater than the pagesize. During the execution of the ON unit or after you return from the ON unit without having emitted a pagemark, the line number may increase indefinitely. However, later evaluation of a LINE format does not cause the ENDPAGE condition, but instead writes a pagemark into the output stream.

The default ON unit places a pagemark into the data stream, resets the linenumber to 1, and returns to the point where the condition was detected.

### The ERROR Condition

ERROR

This condition is signalled when:

- subscripts are out of range and the program was compiled with subscript checking;
- arguments of the SUBSTR built-in function are out of range and the program was compiled with subscript checking;
- storage for based variables is full;
- an error is detected during I/O statement execution;
- an error is detected by the mathematical built-in functions;
- an error is detected by the exponentiate operator;
- as the default response to the ENDFILE condition;

- a conversion error occurs;
- the evaluation of the expression of a DO CASE statement is out of range and an OTHERWISE clause is not given;
- floating-point overflow is detected;
- a floating-point division by zero occurs.

If stack overflow occurs, the ERROR ON unit does not get control. Instead an error message is written on the terminal and execution stops. Any files which were open at the time the overflow occurred are left in an undefined state.

If an ON unit for the ERROR condition attempts to return to the point where the condition was signalled, the program stops.

The default ON unit writes an error message on the user console and executes a STOP statement.

## 9.18 The OPEN Statement

OPEN FILE(file-reference) *[open-option]...;*

where *open-option* may be:

```
DIRECT
INPUT
KEYED
LINESIZE(expression)
OUTPUT
PAGESIZE(expression)
PRINT
RECORD
SEQUENTIAL
STREAM
TITLE(expression)
UPDATE
```

Execution of an OPEN statement associates a data set with a file-control block. The information provided in the open-options is recorded in the file-control block.

Evaluation of the file-reference must yield a FILE value.

The expressions specified in the LINESIZE and PAGESIZE options must yield FIXED BINARY values.

Evaluation of the expression specified in the TITLE option must yield a character-string.

Each open statement can contain only one TITLE, one LINESIZE, one PAGESIZE option, and one FILE option. The LINESIZE option may be specified only for stream data sets. PAGESIZE may be specified only for STREAM OUTPUT PRINT data sets.

If the file is already open, no action is taken and control passes to the statement immediately following the OPEN statement; otherwise, the file is opened as described in Section 8.4.

For example:

```
OPEN FILE(F) TITLE("XYZ.PR") UPDATE KEYED;

OPEN FILE(G);
```

## 9.19 The PROCEDURE Statement

procedurename: PROCEDURE *[(parameter-list)] RETURNS(data type)] [RECURSIVE];*

where *parameter-list* is (identifier *[,identifier]* ...)

The PROCEDURE statement defines the primary entry point of a procedure, specifies parameters for the primary entry point, specifies the attributes of the value returned if the procedure is invoked at the primary entry point as a function reference, and specifies the RECURSIVE attribute if the procedure can be invoked recursively.

The procedurename is a single unsubscripted label which results in the declaration of an entry constant. A GOTO statement cannot transfer control to a PROCEDURE statement.

Each identifier in the parameter-list must identify a level-one variable declared in the procedure headed by this PROCEDURE statement. Furthermore, the number of identifiers in the parameter-list must equal the number of expressions in the argument-list of each CALL statement or function reference that invoked this entry. See Section 5.6 which discusses parameters and arguments.

If control reaches a PROCEDURE statement as a result of completing the execution of the preceding statement, the procedure is skipped. Control is transferred to the statement immediately following the END statement that ends the procedure.

Control enters a procedure only when one of its entry points is invoked by a function reference or by a CALL statement.

If control passes to the PROCEDURE statement by the execution of a CALL statement, the PROCEDURE statement cannot have a RETURNS attribute. Control returns from the procedure to the statement following the call statement by the execution of a RETURN statement which does not contain a return value or by the execution of the END statement.

If control passes to the PROCEDURE statement by the evaluation of a function reference, the PROCEDURE statement must have a RETURNS attribute. Control returns from the procedure to the statement containing the function reference by the execution of a RETURN statement which contains a return value. See Section 5.5 which discusses function references.

Control can leave any procedure by the execution of a nonlocal GOTO statement.

The RETURNS and RECURSIVE attributes can be specified in any order.

For example:

```
P:          PROCEDURE (X,Y,Z);
            .
            .
            .
            END;

PR:         PROCEDURE (A) RETURNS(FIXED DECIMAL(3,2));
            .
            .
            .
            RETURN(B);
            END;
```

## 9.20 The PUT Statement

The PUT statement may take one of the following forms:

PUT *[FILE(file-reference)]* SKIP *[(expression)];*
PUT *[FILE(file-reference)]* PAGE;
PUT *[FILE(file-reference)] [page-control]* EDIT(output-list)(format-list);
PUT *[FILE(file-reference)] [page-control]* LIST(output-list);

where output-list is output-item *[,output-item]....*

An output-item is either an expression, a constant, or (output-list DO increment).

Format-list is described in Section 9.11

*Page-control* is PAGE or SKIP *[(expression)].*

Execution of a PUT statement transmits character-string representations of data to a stream data set.

If the FILE option is not specified, FILE(SYSPRINT) is supplied and is attached to a default operating system filename. If the FILE option is specified, evaluation of the reference must yield a FILE value.

If the file is closed, it is implicitly opened with the STREAM and OUTPUT attributes.

If the PAGE option is given, the file must have the PRINT attribute. The PAGE option is evaluated as if it were a PAGE format. See this chapter.

If the SKIP option is specified, the value of the expression must be a FIXED BINARY integer, k. If the expression is omitted, SKIP(1) is assumed. The data stream is advanced until k linemarks have been output and columnposition is set to one. If k=0, a carriage return character is placed in the data stream, and columnposition is set to one. This enables more than one line of output to be printed on the same line.

Each time control passes to a PUT EDIT or PUT LIST statement, each output-item, beginning with the leftmost output-item, is evaluated, and its value is output to the data stream. If no more output-items remain, control transfers to the statement immediately following the PUT statement.

If the next output-item is an expression, it is evaluated to obtain its value, converted to a character-string according to the conversion rules given in Chapter 7, and output to the data stream.

If the next output-item is a parenthesized output-list followed by a DO increment, the DO increment controls the repetitive evaluation of the output-list as if the parenthesized output-list were a DO group. See this chapter for a description of the DO increment. For an example, see the GET statement.

### PUT LIST

If the next output-item is a scalar, it is selected as the next output value. If the next output-item is an aggregate, each of its scalar components is selected as an output value. The elements of arrays are selected in row-major order and the members of structures are selected in the order in which they are stored in memory.

Once the next output-item is selected, it is converted to a character-string according to the rules for type conversion given in Chapter 7.

If the original output-item was a bit-string value, the resulting character-string is enclosed in quotes and "B" is appended.

If the original output-item was a character-string value and the file does not have the PRINT attribute, the resulting character-string is enclosed in quotes and each contained quote is replaced by a pair of quotes.

Each output-item is placed in the data stream and followed by a blank. For files with the PRINT attribute, each output-item is followed by sufficient blanks to set columnposition to the next tab position. The blank or blanks are always written, even when the output-item happens to fill the line. In that case blanks are inserted up until the second tab stop.

NOTE: Linemarks and pagemarks are inserted into the data stream as determined by the linesize and pagesize of the file-control block.

For example:

```
PUT FILE(F) LIST(Y);
```

Evaluation of this statement causes the data stream identified by F to contain the value of Y.

```
PUT FILE(G) LIST((X(I) DO I = 1 TO 5));
```

Evaluation of this statement causes the data stream identified by G to contain the values of X(1), X(2), X(3), X(4), and X(5) separated by blanks or tabs with linemarks and pagemarks inserted as necessary.

```
PUT FILE(H) LIST(((A(I,J) DO J = 1 TO 10) DO I = 1 TO 5));
```

execution of this statement causes the data stream identified by F to contain the values of A(1,1)...A(1,10),A(2,1)...A(2,10)...A(5,10) separated by blanks or tabs with linemarks inserted as necessary.

### PUT EDIT

The next output-item is established in the same way as in a PUT LIST statement.

Once the output-item is determined, control passes to the format-list. Evaluation of the format-list causes the output value to be converted and placed into the data stream. See the section in this chapter which discusses format-lists.

Example:

```
PUT SKIP EDIT("A=",A) (A(10),E(10,3));
```

If the value of A is 12345678E0, the output will be:

```
A=□□□□□□□□□1.234E+07
```

## 9.21 The READ Statement

READ FILE(file-reference) INTO(reference) *[KEY(expression)];*

The READ statement causes a record to be transmitted from a RECORD INPUT or STREAM INPUT or RECORD UPDATE file to a variable identified by the INTO option.

Evaluation of the reference of the FILE option must yield a FILE value.

If the file is closed, it is implicitly opened with the RECORD, INPUT, and SEQUENTIAL attributes (even if KEY is specified).

The file must have either the INPUT or UPDATE attributes. If the KEY option is specified, the file must also have the KEYED attribute.

If the file is a STREAM FILE, the INTO option must refer to a scalar CHARACTER VARYING variable. The READ statement reads the next input line, assigns it to the variable, sets the current length of the variable, and sets columnposition to linesize. (The effect of the last step is to cause the next input operation to read a new line.)

If the file is a RECORD file, the reference of the INTO option must refer to a scalar or aggregate variable whose storage is connected and aligned on a word or byte boundary. A copy of the currentrecord is assigned to the variable identified by the INTO option.

If the file is a RECORD DIRECT KEYED file, the KEY option must be specified. The expression of the KEY option is converted to a CHARACTER(32) value. This value must be convertible to a FIXED BINARY value called a key. The currentrecord of the file-control block is set to identify the record desginated by the key. If the key is less than or equal to the largest key and if no such record exists, a record of all zero bits is read. if the key is greater than the largest key, the ENDFILE condition is signalled.

If the file is RECORD SEQUENTIAL, the KEY option must not be specified. The record that is read is the one identified by currentrecord. Currentrecord is then set to identify the next record in the data set.

For example:

```
READ FILE(F) INTO(X);
READ FILE(G) INTO(Y) KEY(K);
```

## 9.22 The RETURN Statement

RETURN *[(expression)];*

If the expression is omitted, the RETURN statement terminates the most recent procedure block activation and transfers control to the statement immediately following the CALL statement whose execution created that block activation. If a RETURN is executed in the first (and only) activated procedure block, all open files are closed and execution of the program is terminated. (The effect is the same as executing a STOP statement.)

If the expression is present, the RETURN statement converts the expression to the data type specified in the RETURNS attribute of the ENTRY or PROCEDURE statement used to enter the most recent procedure activation. This converted value is the value of the function reference that invoked the procedure. The current block activation is terminated and control is transferred to continue evaluation of the statement containing the function reference.

For example:

```
RETURN;
RETURN(X + Y);
```

It is an error to attempt to return from a procedure invoked as a function by executing a RETURN statement which does not contain a return value. It is an error to return from a procedure invoked with a CALL statement by executing a RETURN statement that contains a return value.

## 9.23 The REVERT Statement

REVERT condition;

where condition may be ENDFILE(file-reference), ENDPAGE(file-reference), or ERROR.

The REVERT statement reverts (disestablishes) any on-unit established for the specified condition by the current block activation.

If no on-unit is currently established by the current block activation, the REVERT statement has no effect.

For example:

```
A:      PROCEDURE;
        .
        .
        .
        ON ENDFILE(F)
                BEGIN;
                GO TO X1;               /* If ENDFILE(F) occurs GO TO X1*/
                END;

        .
        .
        .
        .
        CALL B;
        .
        .
        .
B:      PROCEDURE;
        .
        .
        .
        ON ENDFILE(F)
                BEGIN;
                GO TO X2;               /* If ENDFILE(F) occurs GO TO X2*/
                .
                END;
        .
        .
        .
        REVERT ENDFILE(F);              /* If ENDFILE(F) occurs GO TO X1*/
        .
        .
        .
        END;
```

## 9.24 The REWRITE Statement

```
REWRITE FILE(file-reference) FROM(reference) KEY (expression);
```

Execution of the REWRITE statement replaces a record in an UPDATE file with the record identified by the FROM option.

Evaluation of the file-reference must yield a FILE value. The file must be open and must have the RECORD, UPDATE, and KEYED attrributes.

The expression of the KEY option is converted to a CHARACTER(32) value. This value must be convertible to a FIXED BINARY value called a key. The currentrecord of the file-control block is set to identify the record designated by the key.

The reference in the FROM option must refer to a scalar or aggregate variable whose storage is connected and aligned on a word or a byte boundary. A copy of the variable identified by the reference replaces the record identified by currentrecord.

For example:

```
REWRITE FILE(G) FROM(Y) KEY(Z);
```

## 9.25 The SIGNAL Statement

```
SIGNAL condition;
```

Execution of the SIGNAL statement causes the most recently established ON unit for the specified condition to be invoked as a procedure. If no ON unit has been established for that condition, the default ON unit is invoked. See Section 9.17 which discusses conditions.

For example:

```
SIGNAL ERROR;
SIGNAL ENDFILE(F);
SIGNAL ENDPAGE(F);
```

## 9.26 The STOP Statement

```
STOP;
```

Execution of the STOP statement closes all open files and terminates execution of the program.

## 9.27 The WRITE Statement

WRITE FILE(file-reference) FROM(reference) *[KEYFROM(expression)];*

Execution of the WRITE statement creates a new record in a RECORD file or writes a line into a STREAM file.

Evaluation of the file-reference must yield a FILE value.

If the file is closed, it is implicitly opened with the RECORD, SEQUENTIAL, and OUTPUT attributes (even if KEYFROM is specified).

The file must not have the INPUT attribute. If the file has the UPDATE attribute, it must also have the KEYED attribute. If the KEYFROM option is specified, the file must have the KEYED attribute and if the file has the KEYED attribute the KEYFROM option must be specified.

If the file is a STREAM file, the FROM option must refer to a scalar CHARACTER VARYING variable. The WRITE statement writes to the file the current value of the variable followed by a linemark. Columnposition is set to one.

If the file is a RECORD file, the FROM option must refer to a scalar or aggregate variable whose storage is connected, aligned on a word or byte boundary. A copy of the variable identified by the reference is the record to be written.

If the file is a RECORD DIRECT KEYED file, the KEYFROM option must be specified. The expression of the KEYFROM option is converted to a CHARACTER(32) value. This value must be convertible to a FIXED BINARY value called the key. The record identified by the FROM option is associated with the key from the KEYFROM option and is transferred to the data set. If the data set already contained a record with this key, it is replaced by the new record. (Note that in Standard PL/I it is an error if a record with this key already exists.)

If the file is RECORD SEQUENTIAL, the KEYFROM option must not be specified. The record identified by the FROM option is output to the data set.

For example:

```
WRITE FILE(F) FROM(OUT1) KEYFROM(K);
```

End of Chapter

# Chapter 10
# Built-In Functions

A *built-in function* is a user-callable function which is an intrinsic part of the PL/I language. A built-in function reference may be used anywhere a user-defined function may be. Built-in function names do not have to be declared to be used. If the name of a built-in function is redeclared in the program, that function cannot be called within the scope of that declaration. However, you may use the built-in function in a contained block if it is redeclared with the BUILTIN attribute. (See Chapter 4.)

A built-in function that has no argument may be used with or without an empty argument list.

The built-in functions may be grouped into the following classes.

1. The arithmetic built-in functions:

| | | | |
|---|---|---|---|
| ABS | FLOOR | MAX | SIGN |
| CEIL | LOG | MIN | SQRT |
| DIVIDE | LOG10 | MOD | TRUNC |
| EXP | LOG2 | ROUND | |

2. The trigonometric built-in functions:

| | | | |
|---|---|---|---|
| ACOS | COS | SIND | TANH |
| ASIN | COSD | SINH | |
| ATAN | COSH | TAN | |
| ATAND | SIN | TAND | |

3. The string built-in functions:

| | | |
|---|---|---|
| BOOL | STRING | VALID |
| COLLATE | SUBSTR | VERIFY |
| INDEX | TRANSLATE | |
| LENGTH | | |

4. The conversion built-in functions:

| | | | |
|---|---|---|---|
| ASCII | CHARACTER | FIXED | RANK |
| BINARY | DECIMAL | FLOAT | UNSPEC |
| BIT | | | |

5. The condition built-in functions:

| | |
|---|---|
| ONCODE | ONFILE |

6. The miscellaneous built-in functions:

| | | | |
|---|---|---|---|
| ADDR | HBOUND | NULL | SIZE |
| DATE | LBOUND | PAGENO | TIME |
| DIMENSION | LINENO | | |

PAGENO, STRING, SUBSTR, and UNSPEC may also be used on the left side of an assignment statement. When used in this way, they are called pseudo-variables. Their use is described in Section 9.1, which describes the assignment statement.

This chapter describes in alphabetical order the general form and purpose of each built-in function. Where applicable, algorithms, remarks, and error conditions are included.

## ABS

### Format

ABS(X)

where X is an arithmetic expression.

### Purpose

Compute the absolute value of X.

### Result Type

Same as X.

### Result Precision

Same as X.

### Algorithm

If X > = 0 then return X.
If X < 0 then return -X.

## ACOS

### Format

ACOS(X)

where X is an arithmetic expression in the range -1 < =X < =1

### Purpose

Returns arc cosine of X: the angle (in radians) whose cosine is X in the range 0 < =ACOS(X) < =PI.

### Result Type

FLOAT BINARY

### Result Precision

Same as X after conversion to FLOAT BINARY.

### Error Condition

If X is not in the range -1 < =X < =1, the error condition is signalled.

## ADDR

### Format

ADDR(X)

where X is a reference to a variable whose storage is connected. X must not be bit aligned.

### Purpose

Returns a pointer that identifies the generation of storage referenced by X.

### Result Type

If X is character, fixed decimal, pictured, or a structure that consists entirely of these data types, the result is a byte pointer; otherwise, it is a word pointer.

### Remarks

The evaluation of X must yield a generation of storage, i.e., X must exist.

```
DECLARE (P,Q) POINTER;
DECLARE 1 X BASED (P),
            2 Y,
            2 Z;
Q= ADDR(X);             /* ERROR: STORAGE IS NOT ALLOCATED */
                        /* FOR X*/
```

It is an error to reference word aligned data with a byte pointer or to reference byte aligned data with a word pointer.

## ASCII

### Format

ASCII(X)

where X is a FIXED BINARY expression.

### Purpose

Convert a FIXED BINARY value into a one-character string.

### Result Type

CHARACTER(1)

### Algorithm

ASCII(X)=SUBSTR(COLLATE,MOD(X,256)+1,1)

### Example

```
DECLARE NUL CHARACTER(1);
NUL=ASCII(0);
```

## ASIN

### Format

ASIN(X)

where X is an arithmetic expression in the range -1 < =X < =1

### Purpose

Returns arc sine of X: the angle (in radians) whose sine is X in the range -PI/2 < =ASIN(X) < =PI/2.

### Result Type

FLOAT BINARY

### Result Precision

Same as X after conversion to FLOAT BINARY.

### Error Condition

If X is not in the range -1 < =X < =1, the error condition is signalled.

## ATAN

### Format

ATAN(X)

where X is an arithmetic expression.

### Purpose

Returns arc tangent of X: the angle (in radians) whose tanget is X in the range -PI/2 < ATAN(X) < PI/2.

### Result Type

FLOAT BINARY

### Result Precision

Same as X after conversion to FLOAT BINARY.

## ATAN(Y,X)

### Format

ATAN(Y,X)

| | |
|---|---|
| where | Y is an arithmetic expression |
| | X is an arithmetic expression |

### Purpose

Returns the angle in radians whose tangent is Y/X.

| | |
|---|---|
| If Y=0, | then<br>if X > 0 returns 0<br>if X < 0 return PI |
| If Y > 0, | then returns 0 < ATAN(Y,X) < PI<br>if X=0 returns (PI/2) |
| If Y < 0, | then returns (-PI) < ATAN(Y,X) < 0<br>if X = 0 returns PI/2) |

### Error Condition

If X=0 and Y=0, the error condition is signalled.

### Result Type

FLOAT BINARY

### Result Precision

The greater of the precision of Y and X, after conversion to FLOAT BINARY.

## ATAND

### Format

ATAND(X)

where X is an arithmetic expression.

### Purpose

Returns arc tangent of X: the angle (in degrees) whose tanget is X in the range -90 < ATAND(X) < 90.

### Result Type

FLOAT BINARY

### Result Precision

Same as X after conversion to FLOAT BINARY.

## ATAND(Y,X)

### Format

ATAND(Y,X)

where Y is an arithmetic expression

X is an arithmetic expression

### Purpose

Returns the angle in degrees whose tangent is Y/X.

| | |
|---|---|
| If Y=0, | then<br>if X > 0 returns 0<br>if X < 0 returns 180 |
| If Y > 0, | then returns 0 < ATAN(Y,X) < 180<br>if X=0 returns 90 |
| If Y < 0, | then returns -180 < ATAN(Y,X) < 0<br>if X = 0, returns -90 |

### Error Condition

If X=0 and Y=0, the error condition is signalled.

### Result Type

FLOAT BINARY

### Result Precision

The greater of the precision of Y and X after conversion to FLOAT BINARY.

## BINARY

### Format

BINARY(X) or BINARY(X,P)

where X is an arithmetic expression or string expression which can be converted to an arithmetic value. If X is FIXED DECIMAL with a nonzero scale factor then P must be given.

P is an integer constant indicating the precision of the result.

### Purpose

Convert X to a BINARY arithmetic value.

### Result Type

If X is FLOAT BINARY, the result is FLOAT BINARY; otherwise, the result is FIXED BINARY.

### Result Precision

If P is omitted, the result has a precision that is determined by the rules for type conversion given in Chapter 7.

### Example

```
DECLARE X FIXED DECIMAL (5,2);
DECLARE Y FIXED BINARY;
Y=BINARY(X,3);
```

## BIT

### Format

BIT(S) or BIT(S,L)

where S is an arithmetic or string expression

L is a positive FIXED BINARY expression.

### Purpose

If L is given, convert S to a bit-string of length L. Otherwise, convert S to a bit-string whose length is determined by the rules for type conversion given in Chapter 7.

### Result Type

BIT

## BOOL

### Format

BOOL(X,Y,Z)

| | |
|---|---|
| where | X is a bit expression. |
| | Y is a bit expression. |
| | Z is a bit-string constant, 4 bits long. |

### Purpose

To define a Boolean function.

### Result Type

Bit-string of length MAX(LENGTH(X),LENGTH(Y))

### Algorithm

If X and Y are null strings, the result is a null string. If X is not the same length as Y, the shorter string is extended on the right with zero bits until X and Y are the same length. Let the bit values within Z be M1,M2,M3,M4 from left to right respectively.

The I-th bit value of the result is set to one of the values M1,M2,M3,M4 depending on the I-th bit value of X and Y according to the following table:

| X(I) | Y(I) | RESULT(I) |
|---|---|---|
| 0 | 0 | M1 |
| 0 | 1 | M2 |
| 1 | 0 | M3 |
| 1 | 1 | M4 |

### Example

The result of BOOL (“1100110”, “0101”, “0110”) is “1001110”.

## CEIL

### Format

CEIL(X)

where X is an arithmetic expression.

### Purpose

Calculate the smallest integer greater than or equal to X.

### Result Type

An integer value of the same type as X.

### Result Precision

If X is FIXED DECIMAL, the precison of the result is (MIN(16,MAX(P-Q+1,1)),0); otherwise, the precision of the result is the precision of X.

### Example

The result of CEIL(-3.1) is -3.
The result of CEIL(3.1) is 4.
The result of CEIL(0) is 0.

## CHARACTER

### Format

CHARACTER(S) or CHARACTER(S,L)

where S is an arithmetic or string expression

L is a positive FIXED BINARY expression

### Purpose

If L is given, S is converted to a character-string of length L; otherwise, S is converted to a character-string whose length is determined by the rules for type conversion given in Chapter 7.

### Result Type

CHARACTER

## COLLATE

### Format

COLLATE( ) or COLLATE

### Purpose

Generates a character-string of length 256 that consists of the set of characters in the ASCII character set in ascending order. The ASCII character set is defined in Appendix H.

### Result Type

CHARACTER(256)

## COS

### Format

COS(X)

where X is an arithmetic expression.

### Purpose

Returns a value which is the cosine of the angle X expressed in radians.

### Result Type

FLOAT BINARY

### Result Precision

Same as X after conversion to FLOAT BINARY.

## COSD

### Format

COSD(X)

where X is an arithmetic expression.

### Purpose

Returns a value which is the cosine of the angle X expressed in degrees.

### Result Type

FLOAT BINARY

### Result Precision

Same as X after conversion to FLOAT BINARY.

## COSH

### Format

COSH(X)

where X is an arithmetic expression.

### Purpose

Returns a value which is the hyperbolic cosine of the angle X expressed in radians.

### Result Type

FLOAT BINARY

### Result Precision

Same as X after conversion to FLOAT BINARY.

## DATE

### Format

DATE( ) or DATE

### Purpose

Returns a character-string which represents the system date. The string is in the form "YYMMDD", where YY,MM, and DD are in the ranges 00:99, 01:12, and 01:31 and represent the year, month, and day respectively.

### Result Type

CHARACTER(6)

## DECIMAL

### Format

DECIMAL(X) or DECIMAL(X,P) or DECIMAL(X,P,Q)

where X is an arithmetic expression or string expression which can be converted to an arithmetic value.

P is an integer constant indicating the precision of the result such that 1 < =P < =16. If P alone is given, Q is assumed to be zero.

Q is an integer constant indicating the scale of the result such that 0 < =Q < =P.

### Purpose

Convert X to a DECIMAL arithmetic value.

### Result Type

FIXED DECIMAL

### Result Precision

If P is omitted, the result has a precision and scale determined by the rules for type conversion given in Chapter 7.

## DIMENSION

### Format

DIMENSION(X,N)

where X is an array variable.

N is a positive integer constant indicating the Nth dimension of X.

### Purpose

Calculate the extent of the Nth dimension of X.

### Result Type

FIXED BINARY

### Result Precision

(15)

### Example

```
DECLARE R FIXED BINARY;
DECLARE A(3:5,2,-10:10,4:7);
.
.
.
R=DIMENSION(A,1);                                   /* R=3 */
.
.
.
```

## DIVIDE

### Format

DIVIDE(X,Y,P) or DIVIDE(X,Y,P,Q)

where X is an arithmetic expression

Y is an arithmetic expression

P is an integer constant indicating the precision of the result.

Q is an integer constant indicating the scale of the result such that 0 < =Q < =P. If X and Y are FIXED BINARY, Q must not be given and is assumed to be zero.

### Purpose

Calculate X divided by Y.

### Result Type

Same as the common arithmetic type of X and Y.

### Result Precision

The precision of the result is (P,Q) or (P).

NOTE: If Y = 0 the program is in error and the results of continued execution are undefined.

## EXP

### Format

EXP(X)

where X is an arithmetic expression.

### Purpose

Returns the value of the base of the natural logarithm, e, raised to the power of X: e**X.

### Result Type

FLOAT BINARY

### Result Precision

Same as X after conversion to FLOAT BINARY.

### Error Condition

Overflow occurs if X > 174.673 (approximately) and the error condition is signalled.

## FIXED

### Format

FIXED(X) or FIXED(X,P) or FIXED(X,P,Q)

where X is an arithmetic expression or string expression which can be converted to an arithmetic value.

P is an integer constant indicating the precision of the result.

Q is an integer constant indicating the scale of the result such that 0 < =Q < =P.

### Purpose

Converts X to a fixed arithmetic value according to the rules for type conversion given in Chapter 7.

### Result Type

If X is FIXED DECIMAL or CHARACTER, the result is FIXED DECIMAL. Otherwise, the result is FIXED BINARY.

### Result Precision

If P or P and Q are given, they are the precision of the result; otherwise, the precision of the result is determined by the rules for type conversion given in Chapter 7.

## FLOAT

### Format

FLOAT(X) or (FLOAT(X,P)

where X is an arithmetic expression or string expression which can be converted to an arithmetic value.

P is an integer constant indicating the precision of the result.

### Purpose

Converts X to a FLOAT arithmetic value according to the rules for type conversion given in Chapter 7.

### Result Type

FLOAT BINARY

### Result Precision

If P is given, it is the precision of the result; otherwise, the precision of the result is determined by the rules for type conversion given in Chapter 7.

## FLOOR

### Format

FLOOR(X)

where X is an arithmetic expression.

### Purpose

Calculate the largest integer that is less than or equal to X.

### Result Type

An integer value of the same type as X.

### Result Precision

If X is FIXED DECIMAL, the precision of the result is (MIN(16,MAX(P-Q+1)),0). Otherwise, the precision of the result is the precision of X.

### Example

The result of FLOOR(3.125) is 3.
The result of FLOOR(-3.125) is -4.
The result of FLOOR(0) is 0.

## HBOUND

### Format

HBOUND(X,N)

where X is an array variable

N is a positive integer constant indicating the Nth dimension of X.

### Purpose

Returns the higher bound of the Nth dimension of X.

### Result Type

FIXED BINARY

### Result Precision

(15)

### Example

```
DECLARE R FIXED BINARY;
DECLARE A(3:5,2,-10:10,4:7);

.
.
.
R = HBOUND(A,1);                              /* R=5 */
.
.
.
R = HBOUND(A,2);                              /* R=2 */
.
.
.
```

## INDEX

### Format

INDEX(S,C)

where S and C are both character expressions or both bit expressions.

### Purpose

Searches a string S for a specified substring C and returns a value indicating the position of the leftmost occurrence of C within S.

### Result Type

FIXED BINARY

### Result Precision

(15)

### Algorithm

If either S or C is a null string, the result is zero. If the substring C is not contained within S, the result is zero; otherwise, the result is an integer indicating the position within S of the leftmost character or bit of the substring C.

### Example

The result of INDEX("ABCDEFG","DEF") is 4.

## LBOUND

### Format

LBOUND(X,N)

where X is an array variable

N is a positive integer constant indicating the Nth dimension of X.

### Purpose

Returns the lower bound of the Nth dimension of X.

### Result Type

FIXED BINARY

### Result Precision

(15)

### Example

```
DECLARE R FIXED BINARY;
DECLARE A(3:5,2,-10:10,4:7);
.
.
.
R=LBOUND(A,1);                    /* R=3 */
.
.
R=LBOUND(A,2);                    /* R=1 */
.
.
.
```

## LENGTH

### Format

LENGTH(S)

where S is either a character or bit expression.

### Purpose

Returns the number of characters or bits in the string S. If S is a VARYING CHARACTER string, LENGTH(S) returns the current length of S.

### Result Type

FIXED BINARY

### Result Precision

(15)

### Remarks

The null string has length 0.

## LINENO

### Format

LINENO(X)

where X is a file value.

### Purpose

Returns the linenumber of the file-control block identified by X.

### Result Type

FIXED BINARY

### Result Precision

(15)

NOTE: X must identify an open file-control block with the PRINT attribute, or the program is in error.

## LOG

### Format

LOG(X)

where X is an arithmetic expression and $X > 0$.

### Purpose

Returns the natural logarithm of X.

### Result Type

FLOAT BINARY

### Result Precision

Same as X after conversion to FLOAT BINARY.

### Error Conditions

If $X < = 0$, the error condition is signalled.

## LOG10

### Format

LOG10(X)

where X is an arithmetic expression and $X > 0$.

### Purpose

Returns the logarithm of X to the base 10.

### Result Type

FLOAT BINARY

### Result Precision

Same as X after conversion to FLOAT BINARY

### Error Conditions

If $X < = 0$, the error condition is signalled.

## LOG2

### Format

LOG2(X)

where X is an arithmetic expression and $X > 0$.

### Purpose

Returns the logarithm of X to the base 2.

### Result Type

FLOAT BINARY

### Result Precision

Same as X after conversion to FLOAT BINARY.

### Error Condition

If $X < = 0$, the error condition is signalled.

## MAX

### Format

MAX(X1,X2)

where X1 and X2 are arithmetic expressions.

### Purpose

Returns the larger argument.

### Result Type

Same as the common arithmetic type of X1 and X2.

## MIN

### Format

MIN(X1,X2)

where X1 and X2 are arithmetic expressions.

### Purpose

Returns the smaller argument.

### Result Type

Same as the common arithmetic type of X1 and X2.

## MOD

### Format

MOD(X,Y)

where X and Y are arithmetic expressions.

### Purpose

Returns the value of x mod y.

### Result Type

The result is a value having the common arithmetic type of X and Y.

### Result Precision

Let (PX,QX) and (PY,QY) represent the precision of X and Y respectively. If the common type of X and Y is FIXED, the precision of the result is:

MIN(16,PY-QY+MAX(QX,QY)),MAX(QX,QY)).

Otherwise, the precision of the result is MAX(PX,PY).

### Algorithm

If Y=0, the result is X; otherwise, the result is X-Y*FLOOR(X/Y)

### Example

The result of MOD(15,4) is 3.
The result of MOD(-15,4) is 1.

## NULL

### Format

NULL() or NULL

### Purpose

Returns a NULL pointer value, a pointer which points to no generation of storage.

### Result Type

POINTER

NOTE: NULL may be used with the INITIAL attribute to initialize a STATIC POINTER variable to NULL.

## ONCODE

### Format

ONCODE( ) or ONCODE

### Purpose

The value returned by this function is the error code of the most recent PL/I runtime error which signalled the ERROR condition. The error codes are listed in Appendix E.

### Result Type

FIXED BINARY

### Result Precision

(15)

## ONFILE

### Format

ONFILE( ) or ONFILE

### Purpose

The value returned by this function is the filename for which the most recent endfile or endpage condition was signalled.

### Result Type

CHARACTER

## PAGENO

### Format

PAGENO(X)

where X is a file value.

### Purpose

Returns the pagenumber of the file-control block identified by X.

### Result Type

FIXED BINARY

### Result Precision

(15)

NOTE: X must identify an open file-control block with the print attribute or the program is in error.

## RANK

### Format

RANK(X)

where X is a character value, 1 character long

### Purpose

Returns an integer representation for an ASCII character.

### Result Type

FIXED BINARY

### Result Precision

(15)

### Algorithm

RANK(X)=INDEX(COLLATE,X)-1

## ROUND

### Format

ROUND(X,K)

where X is an arithmetic expression.

K is a signed integer constant indicating the position within X to be rounded. If K > =0, rounding occurs k digits to the right of the decimal point. If k < 0, rounding occurs -K digits to the left of the decimal point.

### Purpose

Returns X rounded to the specified binary or decimal digit position.

### Result Type

Same as X.

### Algorithm

The result is SIGN(X)*FLOOR(ABS(X)*(B**N)+0.5)/B**N

where B represents the base of X:
B=2 if X is BINARY
B=10 IF X is DECIMAL

If X is FIXED, N=K; otherwise, N=K-E where E is the exponent of X.

### Example

The result 3.215 is returned for ROUND(3.2146,3).

## SIGN

### Format

SIGN(X)

where X is an arithmetic expression.

### Purpose

Returns -1,0, or 1 to indicate the sign of X.

### Result Type

FIXED BINARY

### Result Precision

(15)

### Algorithm

If X $<$ 0, return -1.
If X=0, return 0.
If X $>$ 0, return +1.

## SIN

### Format

SIN(X)

where X is an arithmetic expression.

### Purpose

Returns a value which is the sine of the angle X expressed in radians.

### Result Type

FLOAT BINARY

### Result Precision

Same as X after conversion to FLOAT BINARY

## SIND

### Format

SIND(X)

where X is an arithmetic expression.

### Purpose

Returns a value which is the sine of the angle X expressed in degrees.

### Result Type

FLOAT BINARY

### Result Precision

Same as X after conversion to FLOAT BINARY

## SINH

### Format

SINH(X)

where X is an arithmetic expression.

### Purpose

Returns a value which is the hyperbolic sine of the angle X expressed in radians.

### Result Type

FLOAT BINARY

### Result Precision

Same as X after conversion to FLOAT BINARY.

## SIZE

### Format

SIZE(X)

where X is a simple reference to a level-one variable.

### Purpose

Returns the number of 16-bit words of storage required to allocate a generation of storage for X.

### Result Type

FIXED BINARY

### Result Precision

(15)

### Remarks

If X is a based variable with refer options, SIZE returns a value that depends on the expression contained in the extent expression, not on the reference contained in the refer option.

## SQRT

### Format

SQRT(X)

where X is an arithmetic expression and X > =0.

### Purpose

Returns the positive square root of X.

### Result Type

FLOAT BINARY

### Result Precision

Same as X after conversion to FLOAT BINARY

### Error Conditions

If X < 0, the error condition is signalled.

## STRING

### Format

STRING(S)

where S is a reference to an array or structure whose elements consist entirely of bit- or character-string data. S must be connected and cannot contain the ALIGNED or VARYING attributes.

### Purpose

Returns a scalar string value composed of the concatenation of the values in S.

### Result Type

CHARACTER or BIT, depending on whether S is CHARACTER or BIT.

## SUBSTR

### Format

SUBSTR(S,I,J) or SUBSTR(S,I)

where S is either a BIT or CHARACTER.

I is a FIXED BINARY value indicating the first bit or character of a substring within X.

J is a FIXED BINARY value indicating the length of the substring. If J is not given, the J=LENGTH(S)-I+1.

### Purpose

Return a string which is a copy of a part of the string S starting at the Ith character for a length J.

### Result Type

Same as S.

### Error Conditions

The program is in error if I < 1 or (I+J-1) > length(s) or J > I. If the program is compiled with the subscript-check option these errors signal the error condition; otherwise, these errors will produce unpredictable results.

## TAN

### Format

TAN(X)

where X is an arithmetic expression.

### Purpose

Returns a value which is the tangent of the angle X expressed in radians.

### Result Type

FLOAT BINARY

### Result Precision

Same as X after conversion to FLOAT BINARY.

## TAND

### Format

TAND(X)

where X is an arithmetic expression.

### Purpose

Returns a value which is the tangent of the angle X expressed in degrees.

### Result Type

FLOAT BINARY

### Result Precision

Same as X after conversion to FLOAT BINARY.

## TANH

### Format

TANH(X)

where X is an arithmetic expression.

### Purpose

Returns a value which is the hyperbolic tangent of the angle X expressed in radians.

### Result Type

FLOAT BINARY

### Result Precision

Same as X after conversion to FLOAT BINARY

## TIME

### Format

TIME( ) or TIME

### Purpose

Returns a character-string which represents the time of day. The string is in the format "HHMMSS" where HH, MM and SS are in the ranges 00:23,00:59, and 00:59 and represent hours, minutes, and seconds respectively.

### Result Type

CHARACTER(6)

## TRANSLATE

### Format

TRANSLATE(S,T) or TRANSLATE(S,T,X)

where S is a character expression.

T is a character expression.

X is a character expression. If T is shorter than X, T is padded on the right with blanks unti the length of T is equal to the length of X.

If X is not given, it is assumed to be COLLATE().

### Purpose

The occurrence of a character of X in the string S is replaced by the corresponding character in the string T.

### Result Type

Character-string, of the same length as S.

### Algorithm

If S is the null string, the result is the null string. If S is not the null string, then for each character of S, SI, calculate J=INDEX(X,SI). If J is 0 the corresponding character of the result is SI; otherwise, the corresponding character of the result is the Jth character of T.

### Example

The result of TRANSLATE ("ABCDE.AB","XYZ","ACE.") is "XBYDZ□XB".

## TRUNC

### Format

TRUNC(X)

where X is an arithmetic expression.

### Purpose

Returns the integer portion of X.

### Result Type

A signed integer value of the same type as X.

### Result Precision

If X is FIXED DECIMAL, the precision of the result is (MIN(16,MAX(P-Q+1,1)),0); otherwise, the precision of the result is the precision of X.

### Algorithm

For X < 0, the result is CEIL(X).
For X > =0, the result is FLOOR(X).

### Example

The result of TRUNC(3.125) is 3.
The result of TRUNC(-3.125) is -3.

## UNSPEC

### Format

UNSPEC(X)

where X is a reference to a variable. X may not be a byte-aligned parameter, defined variable, based variable, or subscripted variable.

### Purpose

Returns the internal representation of X.

### Result Type

Bit-string whose length is equal to the length in bits of the generation of storage of X.

## VALID

### Format

VALID(X)

where X is a reference to a scalar pictured value.

### Purpose

The result indicates if the character-string value of X can be edited into the picture declared for X.

### Result Type

Its value is "1"B if the character-string value of X can be edited into the picture declared for X; otherwise, its value is "0"B.

## VERIFY

### Format

VERIFY(S,C)

where S is a character expression.

C is a character expression.

### Purpose

Returns 0 if each of the characters in S occurs in C. Otherwise returns an integer that indicates the leftmost character in S which is not found in C.

### Result Type

FIXED BINARY

### Result Precision

(15)

### Example

```
VERIFY("2A56B","0123456789")
```

Return the value 2 to indicate the position of the first nonnumeric character in the string "2A56B".

End of Chapter

# Chapter 11
# Advice on the Use of PL/I

## 11.1 Programming Style

The difference between a good program and a poor program is frequently a difference in style. This section is intended to provide the novice, as well as the experienced PL/I programmer, guidelines that will enable the programmer to develop a good programming style.

### Modularity

Procedures provide the best mechanism to ensure program modularity. The problem to be programmed should be organized or structured as a set of functional modules, each of which is implemented as a procedure.

A small procedure that is called from within another procedure should be written as an internal procedure. Similarly, a small procedure that is called from within several other small procedures should be written as an internal procedure, as should its callers.

The total size of an external procedure (including its internal procedures) should not exceed 500 lines. External procedures larger than 500 lines are difficult to read. A minor change to any part of the procedure requires recompilation of the entire procedure, thus increasing the time required to make the change.

Although the compiler can compile external procedures of several thousand lines, the number of names, etc., in such procedures may cause the compiler to exceed its available storage. If this occurs, the only solution is to reduce the size of the procedure by partitioning it into two or more external procedures. Since it is often difficult to partition arbitrarily a large procedure, programmers are advised initially to organize a program so that external procedures do not exceed 500 lines.

### Indentation

The single most important factor affecting the readability of a program is the format of the text. It is extremely difficult to read a program that is inconsistently formatted. The following formatting conventions are recommended; but regardless of which conventions are used, all programmers on a given project, team, or administrative unit are urged to adopt the same conventions so that they may readily read each other's code.

a. All executable statements should begin 8 spaces (or 1 tab) from the left magin. This allows labels written in the left margin to be found easily.

b. Each labeled statement should have its label written in the left margin on the line immediately preceding the rest of the statement (which should be indented 8 spaces from the left margin). This makes the label more visible and permits the use of long identifiers as labels, without affecting the alignment of the executable statements.

c. The keywords THEN and ELSE should be indented 3 spaces from their corresponding IF. A DO group written as a then-clause or else-clause should be indented 8 spaces (1 tab) from the corresponding IF. These conventions ensure that corresponding THEN and ELSE keywords are aligned with each other and that the logical nesting of IF statements is visually represented on the page.

d. The executable statements of a DO group (other than a DO group used as a then-clause or else-clause) should be indented 3 spaces from the DO. The END should be aligned with its corresponding DO. These conventions ensure that the logical nesting of DO groups is visually represented on the page.

## Comments

Each procedure should begin with a comment which describes what the procedure does. Each parameter should be defined. A similar comment should exist for each alternate entry point.

The results of the procedure should be clearly described. This includes any arguments passed by reference whose values might change, any global variables which are modified, and the meaning of the expression in any RETURN statements.

Each major section of code should have a descriptive comment.

Each case of a DO CASE statement should begin with a comment of the form:

/* CASE K: description of case */

The END statement of each procedure should be of the form:

END; /* procedure name */

Other comments should be surrounded by sufficient spaces or blank lines to ensure that they are visable.

## DECLARE Statements

Although DECLARE statements can be written anywhere within a procedure, they should all appear immediately following the PROCEDURE statement. Only the forms recommended in Section 4.3 should be used. Because DECLARE statements have no labels, and because they appear before the executable statements, they can begin at the left margin.

## Names

Descriptive names should be used wherever possible, especially for all labels and major variables. Local variables used as DO indexes, pointers, subscripts, etc., should have short, perhaps meaningless, names.

Names should be declared in the innermost block in which they are used. This prevents the accidental use of a variable from within a nested block.

## FORMAT Statements

FORMAT statements should be considered declarations of lines. As such they should appear immediately following any DECLARE statements and they should have meaningful labels such as : PAGE_HEADER, SUBTOTAL_LINE, etc.

## Redundant Parentheses

Redundant parentheses should be used to clarify expressions that contain mixtures of ! and & operators. They also improve the readability of the following:

```
A = B = C;        A = (B = C);
A = -P->X;        A = -(P->X);
```

## ENTRY Statements

ENTRY statements should be used with caution because they introduce the possibility that:

a. A given parameter might not appear in all entries and thus be undefined for some block activations.

b. The return type specified on the entry may differ from that specified in other entries. This is valid but increases run-time space overhead and makes programs difficult to understand.

c. AUTOMATIC and DEFINED variables initialized by assignment statements might be accidentally bypassed when control enters via the ENTRY statement.

## Internal Procedures

Internal procedures should appear immediately following the last executable statement of their containing procedure. This makes the calling procedure occur before the called procedure, and thus most closely approximates the hierarchical relationship among procedures.

An internal procedure should not read or assign to a variable declared in the containing procedures, unless you point this out clearly in a comment near the declaration of the variable. Extensive use of the ability to access variables declared in outer procedures allows each call to an internal procedure to produce numerous implicit results and allows such procedures to have numerous implicit inputs. Both types of access make the program difficult to understand and to modify.

## On-Units

On-units for ENDPAGE or ENDFILE should perform a minimum amount of work before returning to the source of the signal or exiting the on-unit by means of a GOTO.

On-units entered for the ERROR condition cannot return to the source of the signal. An attempt to do so will result in program termination.

## Abbreviations

Use abbreviations consistently. Do not write CHARACTER sometimes and CHAR other times.

Statement keywords should not be abbreviated. Attribute keywords that are long and frequently used can be abbreviated.

## %INCLUDE and %REPLACE

All array sizes should be defined by means of %REPLACE statements:

```
%REPLACE TABLE_SIZE BY 25;
DECLARE TABLE(TABLE_SIZE) FIXED INTERNAL STATIC;
DO K = 1 TO TABLE_SIZE;
.
.
.
```

All external variables, files, and named constants, that are used in more than one external procedure should be declared in a file that is inserted in all appropriate places by means of a %INCLUDE statement.

## An Example of Program Style

```
EXAMPLE:
        PROCEDURE(A,B) RETURNS(POINTER);

/* THIS PROCEDURE DOES NOTHING EXCEPT DEMONSTRATE THE PROGRAMMING
   STYLE RECOMMENDED IN SECTION 11.1 */

DECLARE PAYROLL FILE;
DECLARE (A,B) FIXED BINARY;
DECLARE P POINTER STATIC INITIAL (NULL);
DECLARE NEXT POINTER;

DECLARE 1 DEPT_RECORD BASED,
        2 NAME CHARACTER(10),
        2 TAG FIXED DECIMAL(7,2);

START:

        IF A = B
                THEN DO;
                        PUT LIST(A,B);
                        CALL FOO(A);
                        END;
                ELSE IF A < B
                        THEN P = ADDR(A);
                        ELSE DO;
                                A = B + 7;
                                P = NULL;
                                END;
DO WHILE (P = NULL);
        P -> DEPT_RECORD.NAME = "ABC";
        A = B
        P = P -> NEXT;
END;
/* THIS IS A LARGE COMMENT ABOUT THE ACTION TO FOLLOW */

DO A = 1 TO 10;
        DO B = 5 TO 7;
                P -> DEPT_RECORD.TAG = A +B;
                WRITE FILE(PAYROLL) FROM (P->DEPT_RECORD);
        END;
END;

RETURN(P);

FOO:
        PROCEDURE(X);

DECLARE X FIXED BINARY;

        PUT LIST("MESSAGE");
        END; /* FOO */

        END; /* EXAMPLE */
```

## 11.2 Code Efficiency

### Input/Output

Record I/O is faster and requires less runtime space than stream I/O. In programs where these efficiencies are important, record I/O should be used instead of stream I/O. A STATIC structure consisting entirely of CHARACTER and PICTURE fields can serve as a declaration of a line. A PUT operation is programmed by assigning values to the appropriate fields and then writing the structure using a WRITE statement. The reverse procedure can be used to program a GET operation. Output fields that are constant can be given their values by means of INITIAL attributes.

The runtime support routines for stream I/O are loaded as required by each program. Four sets of routines are loaded, one each for: GET LIST, GET EDIT, PUT LIST, and PUT EDIT. Both GET EDIT and PUT EDIT load a common format scanner.

### Begin Blocks

Begin blocks should be used only as ON units, and when necessary to control allocation and scoping of variables. A begin block should not be used as a case of a DO CASE statement, unless to control storage allocation. Use a DO group instead.

### Arithmetic

Binary arithmetic is always faster and requires fewer generated machine instructions than does arithmetic on decimal or pictured data. This is particularly true on machines without the hardware commercial instruction set.

FIXED DECIMAL constants such as 7.5 should be written as 7.5E0 if used in a floating-point expression. Likewise, 7. should always be written as 7, without a decimal point.

### Variable Size Data

To obtain the most efficient generated code, whenever possible the sizes of arrays and the lengths of strings should be declared with an integer constant. However, AUTOMATIC, DEFINED, and BASED variables may have variable sizes, and parameters may have * sizes.

Whenever possible, arrange the members of a structure so that all constant size members precede any variable size members.

Remember that any variable size given for an AUTOMATIC or DEFINED variable must be computable upon entry to the block in which the AUTOMATIC or DEFINED variable is declared. That is, the size variable or expression must not refer to another AUTOMATIC or DEFINED variable declared in the same block.

Also, remember that the size of a BASED variable is evaluated every time the variable is referenced, whereas the sizes of AUTOMATIC or DEFINED variables are evauated upon block entry and are saved in the stack frame of that block activation. Therefore, use simple variables rather than expressions as sizes in BASED variables.

## 11.3 Program Interchange

If a program is to be compiled by another implementation of PL/I, the following should be observed:

a. Names must not exceed 31 characters.

b. All arithmetic variables should be fully declared; that is they cannot rely on defaults. Write: FIXED BINARY(p) or FIXED DECIMAL(p,q) or FLOAT BINARY(p).

c. Do not use the DO CASE or %REPLACE statements.

d. Do not use the UNSPEC, SIZE, ASCII, or RANK built-in functions.

e. Do not use READ and WRITE statements for stream files.

f. Do not use the special features of interactive console I/O which are described in the appendixes.

Because Data General PL/I is a subset of ANSI Standard PL/I, it does not support all features found in some implementations of PL/I. If a program developed elsewhere is to be compiled by Data General PL/I, you should expect to make numerous modifications to the original program. The following items are provided as a convenient check list of most differences between ANSI standard and Data General PL/I.

a. The DEFAULT and LOCATE statements are not supported.

b. The following attributes are not supported: AREA, COMPLEX, CONDITION, CONTROLLED, ENVIRONMENT, FORMAT, GENERIC, LIKE, LOCAL, OFFSET, OPTIONS, POSITION, and UNALIGNED.

c. The following attributes are supported, but their keywords have no meaning and are not permitted within the text of a program: CONSTANT, MEMBER, NONVARYING, PARAMETER, PRECISION, REAL, STRUCTURE, UNALIGNED.

d. The following attributes may occur only in an OPEN statement: DIRECT, INPUT, KEYED, OUTPUT, PRINT, RECORD, SEQUENTIAL, STREAM, and UPDATE.

e. All arithmetic variables must be declared as FIXED BINARY(p), FIXED DECIMAL(p,q), or FLOAT BINARY(p). Any occurrence of FLOAT DECIMAL must be changed to FLOAT BINARY(p).

f. All names must be declared either in a DECLARE statement or by appearing as a label.

g. Entry points to an external procedure A called from within another external procedure B, must be declared by DECLARE statements written in B. No other entry points can be declared by DECLARE statements.

h. All expressions must be scalar valued, except for assignments of the form A = B; where A and B are arrays or structures of identical size, shape, and component data types. Aggregate promotion is not supported.

i. The following built-in functions are not supported: ADD, AFTER, ALLOCATION, BEFORE, COMPLEX, CONJG, COPY, DECAT, DOT, EMPTY, ERF, ERFC, EVERY, HIGH, IMAG, LOW, MULTIPLY, OFFSET, ONCHAR, ONFIELD, ONKEY, ONLOC, ONSOURCE, POINTER, PRECISION, PROD, REAL, REVERSE, SOME, SUBTRACT, SUM.

j. The MIN and MAX built-in functions take only two arguments.

k. The extent expressions on the RETURNS attribute must be unsigned integer constants.

l. The ALLOCATE statement requires the SET option. The IN option is not supported.

m. The DO statement cannot contain the REPEAT option, nor can it contain both the WHILE and TO options.

n. The END statement may not contain a label. Multiple closure is not supported.

o. The GET and PUT statements cannot contain the COPY, STRING, or DATA options.

p. RECORD KEYED SEQUENTIAL I/O is not supported.

q. The character pair /* may not be embedded in a comment.

r. All functions must yield scalar values.

s. Condition prefixes are not supported.

t. The picture character A, E, I, K, R, T, X, and Y are not supported.

u. Assignment BY NAME is not supported.

v. The assignment statement can only have one target.

w. CONTROLLED storage is not implemented. Most uses of CONTROLLED storage can be easily replaced by the use of BASED storage.

x. The INITIAL attribute can only be used on STATIC variables. Use assignment statements to initialize AUTOMATIC or BASED variables.

y. The only conditions supported are: ERROR, ENDFILE, and ENDPAGE. Most conditions found in other implementations of PL/I are subcases of the ERROR condition. User-defined conditions are not supported.

z. Format lists cannot contain expressions or variables.

aa. All ON units must contain a begin block. The SNAP and SYSTEM options are not supported.

For example:

```
ON ENDFILE(F)
          BEGIN;
          .
          .
          .
          END;
```

## 11.4 Common Programming Errors

Most source program errors result in a meaningful compile-time diagnostics. A few errors result in strange or inappropriate compile-time messages. Some errors cannot be detected at compile-time and the cost of detecting them at runtime would be prohibitive. The following list of errors are either not detected, or produce misleading diagnostics.

1. Failure to end a comment with */. (This causes all text up to the next */ to be considered part of the comment.)

2. Failure to enclose string constants in quotes ‘‘, and failure to use pairs of quotes within a character-string constant. (This causes program text to be considered part of the string constant, or may cause part of the string constant to be considered as program text.)

3. Replacing a keyword by means of a %REPLACE statement.

4. Declaring an external entry point in a manner that disagrees with the actual number and types of its parameters.

5. Using a pointer value that was derived by allocating or taking ADDR of a byte-aligned variable to access a word-aligned variable or vice versa.

6. Accessing a BASED variable with a null or otherwise invalid pointer value.

7. Using an invalid subscript value or invalid argument to SUBSTR built-in function or pseudo-variable. The /SUB option causes the generated code to detect this error and signal the ERROR condition.

8. Calling a procedure recursively without writing RECURSIVE in its PROCEDURE statement.

9. Accessing a variable that has not had a value assigned to it.

10. Using a BASED variable of type t to access the same storage as another variable whose type is not t. This can produce errors if optimization of level greater than 1 is requested.

11. Computing a FIXED BINARY value that excees 15 bits of precision sometimes produces a negative result or a MOD 32768 result.

End of Chapter

# Appendix A Keywords

| Keyword | Abbreviation | Use of Keyword |
|---|---|---|
| A | | format item |
| ABS | | built-in function |
| ACOS | | built-in function |
| ADDR | | built-in function |
| ALIGNED | | data attribute |
| ALLOCATE | ALLOC | statement |
| ASCII | | built-in function |
| ASIN | | built-in function |
| ATAN | | built-in function |
| ATAND | | built-in function |
| AUTOMATIC | AUTO | data attribute |
| B | | format item |
| B1 | | format item |
| B2 | | format item |
| B3 | | format item |
| B4 | | format item |
| BASED | | data attribute |
| BEGIN | | statement |
| BINARY | BIN | data attribute, built-in function |
| BIT | | data attribute, built-in function |
| BOOL | | built-in function |
| BUILTIN | | data attribute |
| BY | | option of DO statement |
| CALL | | statement |
| CASE | | option of DO statement |
| CEIL | | built-in function |
| CHARACTER | CHAR | data attribute, built-in function |
| CLOSE | | statement |
| COLLATE | | built-in function |
| COLUMN | COL | format item |
| COS | | built-in function |
| COSD | | built-in function |
| COSH | | built-in function |
| DATE | | built-in function |
| DECIMAL | DEC | data attribute, built-in function |
| DECLARE | DCL | statement |
| DEFINED | DEF | data attribute |
| DELETE | | statement |
| DIMENSION | DIM | data attribute, built-in function |
| DIRECT | | file attribute |
| DIVIDE | | built-in function |
| DO | | statement |
| E | | format item |
| EDIT | | option of GET and PUT statements |
| ELSE | | optional clause of IF statement |
| END | | statement |
| ENDFILE | | condition |
| ENDPAGE | | condition |
| ENTRY | | statement, data attribute |
| ERROR | | condition |
| EXP | | built-in function |
| EXTERNAL | EXT | data attribute |
| F | | format item |
| FILE | | data attribute, option of GET and PUT statements, requirement of OPEN, CLOSE, READ, WRITE, REWRITE, and DELETE statements |
| FIXED | | data attribute, built-in function |

| | | |
|---|---|---|
| FLOAT | | data attribute, built-in function |
| FLOOR | | built-in function |
| FORMAT | | statement |
| FREE | | statement |
| FROM | | requirement of WRITE and REWRITE statements |
| GET | | statement |
| GOTO | GO TO | statement |
| HBOUND | | built-in function |
| IF | | statement |
| %INCLUDE | | source modification statement |
| INDEX | | built-in function |
| INITIAL | INIT | data attribute |
| INPUT | | file attribute |
| INTERNAL | INT | data attribute |
| INTO | | requirement of READ statement |
| KEY | | option of READ statement, requirement of REWRITE and DELETE statements |
| KEYED | | file attribute |
| KEYFROM | | option of WRITE statement |
| LABEL | | data attribute |
| LBOUND | | built-in function |
| LENGTH | | built-in function |
| LINE | | format item |
| LINENO | | built-in function |
| LINESIZE | | file attribute |
| LIST | | option of GET and PUT statements |
| LOG | | built-in function |
| LOG10 | | built-in function |
| LOG2 | | built-in function |
| MAX | | built-in function |
| MIN | | built-in function |
| MOD | | built-in function |
| NULL | | built-in function |
| ON | | statement |
| ONCODE | | built-in function |
| ONFILE | | built-in function |
| OPEN | | statement |
| OTHERWISE | | optional clause of DO CASE statement |
| OUTPUT | | file attribute |
| P | | format item |
| PAGE | | format item, option of PUT statement |
| PAGENO | | built-in function, pseudo-variable |
| PAGESIZE | | file attribute |
| PICTURE | PIC | data attribute |
| POINTER | PTR | data attribute |
| PRINT | | file attribute |
| PROCEDURE | PROC | statement |
| PUT | | statement |
| R | | format item |
| RANK | | built-in function |
| READ | | statement |
| RECORD | | file attribute |
| RECURSIVE | | option of PROCEDURE statement |
| REFER | | option of BASED attribute |
| %REPLACE | | source modification statement |
| RETURN | | statement |
| RETURNS | | data attribute, option of PROCEDURE and ENTRY statements |
| REVERT | | statement |
| REWRITE | | statement |
| ROUND | | built-in function |
| SEQUENTIAL | SEQ | file attribute |
| SET | | requirement of ALLOCATE statement |
| SIGN | | built-in function |
| SIGNAL | | statement |
| SIN | | built-in function |

| | | |
|---|---|---|
| SIND | | built-in function |
| SINH | | built-in function |
| SIZE | | built-in function |
| SKIP | | format item, option of GET and PUT statements |
| SQRT | | built-in function |
| STATIC | | data attribute |
| STOP | | statement |
| STREAM | | file attribute |
| STRING | | built-in function, pseudo-variable |
| SUBSTR | | built-in function, pseudo-variable |
| SYS | | built-in function |
| SYSIN | | default input file constant |
| SYSPRINT | | default output file constant |
| TAB | | format item |
| TAN | | built-in function |
| TAND | | built-in function |
| TANH | | built-in function |
| THEN | | required clause of IF statement |
| TIME | | built-in function |
| TITLE | | file attribute |
| TO | | option of DO statement |
| TRANSLATE | | built-in function |
| TRUNC | | built-in function |
| UNSPEC | | built-in function, pseudo-variable |
| UPDATE | | file attribute |
| VALID | | built-in function |
| VARIABLE | | data attribute |
| VARYING | VAR | data attribute |
| VERIFY | | built-in function |
| WHILE | | option of DO statement |
| WRITE | | statement |
| X | | format item |

End of Appendix

# Appendix B
# Symbols

This appendix lists the symbols that you use to make up a PL/I program. Some of the symbols are made up of multiple characters (for example, */ or < =). These symbols are called compound symbols; you may not put spaces or any other characters between the characters making up a compound symbol.

## Symbols Used in Identifiers

| | |
|---|---|
| A through Z | used to compose declared names and keywords. |
| a through z | used to compose declared names and keywords. Keywords may be made up of any combination of capital and small letters (e.g., BeGIn), but capital and small letters represent different declared names (e.g., a and A may describe totally different quantities). |
| 0 through 9 | used to compose declared names, but these digits may not be the first character of a declared name. |
| _(underscore) | used to compose declared names, but must not be a declared name's first character. |

## Symbols Used in Numeric Constants

| | |
|---|---|
| 0 through 9 | used to compose numeric constants |
| . | period: used as a decimal point in numeric constants. |
| E | used to separate the exponent in a numeric constant. |
| + | used to indicate a positive exponent. |
| - | used to indicte a negative exponent. |

## Symbols Used in Bit-String Constants

| | |
|---|---|
| 0 through 9 | used to indicate the value of a bit-string constant. |
| A through F | used to indicate the value of a bit-string constant. |
| B,B1,B2,B3,B4 | used to indicate the representation of a bit-string constant. |

## Symbols Used in Picture Definitions

| | |
|---|---|
| 9 | digit specifier |
| V | decimal point specifier |
| Z | zero-suppression character |
| * | zero-suppression and asterisk insertion character |
| , | comma: insertion character |
| . | period: insertion character |
| / | slash: insertion character |
| B | blank insertion character |
| $ | dollar sign specifier |
| S | sign specifier |
| + | plus sign specifier |
| - | minus sign specifier |
| CR | credit specifier |
| DB | debit specifier |

## Operator Symbols

| | |
|---|---|
| + | arithmetic operation (addition), prefix operator (plus) |
| - | arithmetic operator (subtraction), prefix operator (minus) |
| * | arithmetic operator (multiplication). |
| / | arithmetic operator (division). |
| ** | arithmetic operator (exponentiation). |
| ^ | bit-string prefix operator (complement) |
| & | bit-string operator (and) |
| ! | bit-string operator (inclusive or) |
| !! | string operator (concatenation). |
| = | relational operator (equality). |
| ^= | relational operator (inequality). |
| < | relational operator (less than). |
| ^< | relational operator (not less than). |
| > | relational operator (greater than). |
| ^> | relational operator (not greater than). |
| < = | relational operator (less than or equal to). |
| > = | relational operator (greater than or equal to). |
| = | assignment operator. |

## Bracketing Symbols

| | |
|---|---|
| () | parenthesis: used to enclose subscripts, argument and parameter lists, I/O lists, and FORMAT lists. Required by various options of the ALLOCATE, DECLARE, DO, ENTRY, ON and PROCEDURE statements, and the I/O statements. Used to define the order of evaluation of parts of expression. |
| ” ” | Used to bracket string constants and picture definitions. |
| /* */ | Used to bracket comments. |

## Miscellaneous Symbols

| | |
|---|---|
| . | period: structure qualification. |
| -> | pointer qualification. |
| , | comma: used to separate items in a list. |
| ; | semicolon: used to terminate statements. |
| : | colon: used to separate labels from the statements they label, and to separate upper from lower dimension bounds in an array declaration. |
| % | used to begin keywords which identify source-text modification statements. |

End of Appendix

# Appendix C
# Using PL/I under AOS

## Compiling PL/I Programs

### Command Line

The compiler is invoked by executing a CLI command of the form:

```
XEQ PL1 filename
```

where filename or filename.PL1 is the name of an AOS file that contains the source text of a single external procedure. The compiler will search for and compile filename.PL1 or, if that file does not exist, filename. If filename already ends in .PL1, the compiler will look only for that name.

The output of the compiler is an object module and, optionally, a listing file. Unless specified otherwise, the object module is written to a file named filename.OB, where filename is the name of the file being compiled without the ending .PL1.

All compilation error messages will be written to @ OUTPUT, unless they are directed elsewhere.

The following global switches may be used to specify compiler options:

| | |
|---|---|
| /E=name | Send compiler error messages to name instead of @ OUTPUT. If name exists, it will be appended to. |
| /L | Append a listing to the current @ LIST file. The listing consists of line-numbered source text, a variable map, any compilation error messages, and compilation statistics. |
| /L=name | Write a listing to file name. If name exists, it will be appended to. |
| /N | Do not produce an object file. |
| /NEST | Print the nesting level of blocks and groups on the source listing. |
| /O=name | Write the object file to file name. |
| /OPT | Compiler optimization level 3. See the discussion of optimization later in this appendix. |
| /OPT=1 | Compiler optimization level 1. |
| /OPT=2 | Compiler optimization level 2. |
| /OPT=3 | Compiler optimization level 3. |
| /STAT | Write compilation statistics to @ OUTPUT. |
| /SUB | Compile code into the program which will check for out-of-bounds subscripts and arguments to SUBSTR. |

### Examples:

```
XEQ PLI/L PLAN
```

Compile PLAN.PL1 or, if not found, PLAN. Object file is PLAN.OB. Write a listing to @ LIST.

```
X PL1/L=BUDGET.LS/SUB BUDGET.PL1
```

Compile BUDGET.PL1. Include code for subscript and SUBSTR checking. Write a listing to BUDGET.LS.

```
X PL1/O=NEWMED.OB/OPT MED
```

Compile MED.PL1 or MED. Object code is output to NEWMED.OB. Fully optimize code.

## Optimization

If you use /OPT on the PL/I command line, the optimization phase of the compiler will be invoked during compilation. Optimization can occur on three levels.

Level 1: Boolean expressions are optimized so that once the result is known, the rest of the expression is not evaluated.

For example: IF A=10 ! C=D THEN DO;

If A=10, there is no need to evaluate the rest of the expression.

GOTO and branch code is improved.

Level 2: Level 1, plus redundant computations are eliminated without recording any code.

Level 3: Level 2, plus invariant computations are removed from loops.

This level of optimization cannot be used if any DO group contains code that meets all three of these conditions:

1. **Does not depend on variables set within the group.**
2. **Would be unconditionally executed if the DO group was executed.**
3. **Must not be executed unless the DO group is executed.**

For example:

```
A=10;
DO WHILE(X);
.
.
.
A=B;
END;
```

**If A and B are not used elsewhere in the group, level 3 optimization would move A=B; ahead of the group. If A=B; must not be executed unless the DO group is, you cannot use level 3 optimization.**

## Macros

We suggest that you write one or more CLI macros to facilitate compilation.

For example, a macro called PL1LS.CLI might contain:

```
PUSH
SEARCHLIST :PL1,[!SEARCHLIST]
XEQ PL1/L=%1%.LS%0\L% %1%
POP
```

The command PL1LS name will now push a CLI level, add :PL1 to your searchlist, compile name.PL1 and produce a listing called name.LS, and pop to your original CLI state. Any switch you use on PL1LS except /L will be used on the PL1 command.

For details of writing CLI macros, see the AOS Command Line Interpreter User's Manual (093-000122).

## Compiler Files

The compiler consists of the following files:

| | |
|---|---|
| PL1.PR | Compiler phase 1 root |
| PL1.OL | Compiler phase 1 overlays |
| PL1GEN.PR | Compiler phase 2 root |
| PL1GEN.OL | Compiler phase 2 overlays |
| PL1DRIVER | Language syntax tables |

You must have access to these files to compile a PL/I program.

The compiler produces the following temporary work files which are deleted upon termination of the compiler. If the compiler terminates abnormally, some of them may not be deleted.

?(process ID no.).PL1.TOKEN.TMP
?(process ID no.).PL1.MACRO.TMP
?(process ID no.).PL1.OPT.TMP
?(process ID no.).PL1.SPILL.TMP
?(process ID no.).PL1.ERROR.TMP

A different set of work files is used for each process using the compiler.

## Compile-Time Diagnostics

A diagnostic message is written to the error and listing files for each error diagnosed by the compiler. The message consists of an error number, a severity level, the source line number where the error begins, and a description of the error. Appendix D contains a list of the error messages.

There are four severity levels of errors:

| | |
|---|---|
| Severity 1: | Warning only. The program is valid and produces correct object code. The compiler will give a warning for minor differences between Data General PL/I and ANSI Standard PL/I. A warning will also be issued if you use a feature in a way that will give results other than what you might expect. |
| Severity 2: | The program contains an invalid statement, but the compiler will probably produce working object code. |

| | |
|---|---|
| Severity 3: | The program is invalid and the generated code will not work. |
| Severity 4: | The program is invalid and compilation cannot continue. The compiler aborts on a severity 4 error. |

An error message that begins "COMPILER ERROR:" may be produced by compiling an invalid program. The compilation should also produce one or more genuine diagnostic messages. If the only diagnostic issued for a compilation begins with "COMPILER ERROR:", please copy the source program and forward it to your Data General representative.

## Binding PL/I Programs

### The PL/I Bind Macro

The AOS binder utility is used to bind your object files into an executable .PR file. To simplify the binder command line, a macro called PL1BIND.CLI is supplied with the PL/I compiler. To bind your PL/I program, type

PL1BIND mainobject *[object...]*

where mainobject is the PL/I object program in which execution is to begin, and each object is an external procedure produced either by the PL/I compiler or the AOS assembler. The .OB ending of the object files need not be typed.

The binder will produce an executable file called mainobject.PR.

The PL1BIND macro will accept the switches of the AOS BIND command. For example, /P=name can be used to give a different name to the .PR file. /L or /L=name can be used to get a detailed binder listing. For details on using the binder, see the AOS Binder Use's Manual (093-000190).

### Program Libraries

PL1.LB is a library of all the PL/I runtime routines required for execution of PL/I programs. The PL1BIND command will cause PL1.LB to be searched for any routines needed by a PL/I program. Examples of the types of programs in PL1.LB include mathematical built-in functions, conversion routines, and I/O routines.

To simplify the PL1BIND command line, the PL/I user can build his own program libraries. For details, see the AOS Library File Editor User's Manual (093-000198).

Note that declaring an external ENTRY constant in a PL/I program will not cause a program with that entry name to be loaded from a library. Only the actual use of the name in a CALL statement or function reference will cause the program to be loaded from a library.

This feature allows you to %INCLUDE into a program ENTRY constant declarations for all the procedures in a library. Only the procedures that the program uses will be bound in with it.

### Binding for Different Computer Models

If a program uses DECIMAL, PICTURE, or CHARACTER data, it will be a little smaller and execute faster if it is bound with PL1ECIS.OB. This can only be done if the program is to be run exclusively on C SERIES ECLIPSE computers. To bind with PL1ECIS.OB, include PL1ECIS at the end of the PL1BIND command line. If you want to always bind with PL1ECIS.OB, you can edit PL1ECIS into the PL1BIND.CLI macro, just ahead of PL1.LB.

A program which uses CHARACTER data will execute faster on an S/130 ECLIPSE which has the character instruction set if it is bound with PL1CHIS.OB. A program bound with PL1CHIS.OB will run only on an S/130 with the character instruction set, or on a C SERIES ECLIPSE.

Using PL1ECIS or PL1CHIS enables the additional instructions on a C SERIES or S/130 ECLIPSE to be used. Software emulators for these instructions will be loaded from PL1.LB for programs bound without PL1ECIS or PL1CHIS.

### Binder Error

There are three situations not diagnosed by the PL/I compiler which will cause the binder to issue the error message: ATTEMPT TO OVERWRITE DATA.

1. Attempt to initialize one STATIC EXTERNAL variable to different values from different procedures.
2. Duplicate use of a label array constant; that is, using the same subscript on the same label name within a block.
3. Attempt to initialize two unconnected arrays which share words of storage.

If you attempt to bind together object files which contain two external entry constants with the same name, the binder will give the message:

```
MULTIPLY DEFINED SYMBOL symbolname IN MODULE objectname
```

If a program which calls an external procedure is bound without that procedure, the binder will give two errors:

```
SYMBOL UNDEFINED AT END OF PASS 1 symbolname
ATTEMPT TO RELOCATE DATA FROM UNDEFINED SYMBOL
```

For details on these and other binder errors, see the *AOS Binder User's Manual* (093-000190).

## Executing PL/I Programs

To execute the .PR file which the binder produced, type the CLI command:

X *[/S]* name

Name is the name of the .PR file. The .PR suffix does not have to be typed.

### Runtime Error Messages

The default on-units for the ERROR and ENDFILE conditions write an error message to @ OUTPUT and execute a STOP statement. For example, if an attempt is made in procedure GET_DATA to open a file for input, and the file does not exist, the default on-unit will write the following to @ OUTPUT:

```
ERROR CONDITION 8 SIGNALLED FROM GET_DATA
SYSTEM ERROR ON ATTEMPTED OPEN
FILE DOES NOT EXIST
PL/I ERROR
```

The first line of the message gives the ONCODE of the error condition, and gives the name of the procedure in which the error occurred. The second line gives the text of the error messages. For a list of ONCODES and error messages, see Appendix E.

If the second line begins with the words SYSTEM ERROR, the error occurred as a result of an AOS system call. Only in this case will the third line of the error message give the AOS system error message.

Finally, the last line of the message is always PL/I ERROR. However, if the /S switch is used on the XEQ command, PL/I ERROR will not be written to @ OUTPUT, but instead the CLI STRING variable will be set equal to PL/I ERROR. If the /S switch is used and the program ends without an error, the CLI STRING variable will be set to the null string. For details on the XEQ command and STRING, see the AOS Command Line Interpreter User's Manual (093-000122).

## PL/I Data Sets in AOS

### AOS Record Types

The different PL/I file types require different AOS record types. When PL/I creates an AOS data set, it automatically gives it the proper AOS record type; when you use AOS data sets as INPUT and UPDATE files in your programs, you must be sure that the data set has the right record type for the PL/I file to which it is attached. The following table gives the record types required by the PL/I file types:

| PL/I File Type | AOS Record Type |
|---|---|
| STREAM | data sensitive |
| RECORD SEQUENTIAL | variable |
| RECORD DIRECT | 256-word fixed-length record |

The first record of an AOS fixed-length record file is record 0. Therefore, the first record of a RECORD DIRECT data set is accessed with a key value of 0.

### Special Properties of Interactive I/O to the User Console

AOS has two special generic files, @INPUT and @OUTPUT, which are used for I/O to the user console. If these files are opened for STREAM INPUT and STREAM OUTPUT in a PL/I program, they are given several special features which simplify interactive I/O:

1. Input and outut are synchronized on @INPUT and @OUTPUT so that the value of the columnposition for both data sets corresponds to the position of the cursor on the console screen.

2. When you use GET EDIT with @INPUT, GET EDIT removes the linemark ending the input and pads the input with blanks until it is 80 characters long. This has two consequences:

   a. If the input is shorter than the GET EDIT data format item which reads it, the blanks will keep the statement from raising the endfile condition.

   b. If you use several separate GET EDIT statements to read data from @INPUT, each statement after the first must have the SKIP keyword or the SKIP format item. Otherwise, the statement will simply read the blanks which were used to pad the input for the first GET EDIT statement.

3. When a GET LIST statement reads a line from @INPUT, it replaces the linemark with a blank. Consequently, the user need not insert a terminator after the last input item.

4. When a PUT LIST statement writes data to @OUTPUT, the data appears on the screen as soon as the PUT LIST statement has been executed. With other data sets, the data is written only when the line is full.

5. You can use CTRL D to signal ENDFILE from @INPUT.

6. If @OUTPUT is opened as a PRINT file, the pagesize is infinite. ENDPAGE will never be signalled.

### AOS Generic Files

If a PL/I program opens file constants SYSPRINT and SYSIN without specifying a TITLE option, the file constants will be attached to the AOS generic files @LIST and @DATA. These files must be set to valid AOS files by use of the CLI commands LISTFILE and DATAFILE.

When @LIST is opened for output, it is not first deleted as are most OUTPUT files. Instead, the output will be appended to the current @ LIST file.

End of Appendix

# Appendix D
# Compiler Error Messages

The following are diagnostic messages issued by the compiler. The first number before each message is the message number. The second number is the error's severity level.

Whenever a $ occurs in the text of a message, the $ will be replaced by an identifier name or picture declaration when the message is issued by the compiler.

```
 1 2     THE LOGICAL END OF THE EXTERNAL PROCEDURE IS FOLLOWED BY ADDITIONAL
         TEXT.  THE TEXT HAS BEEN IGNORED.

 2 2     INSUFFICIENT END STATEMENTS. AN ADDITIONAL END STATEMENT HAS
         BEEN SUPPLIED.

 3 3     THIS IS AN UNRECOGNIZABLE STATEMENT.

 4 3     A FORMAT STATEMENT MUST BE LABELED WITH ONE UNSUBSCRIPTED LABEL.

 5 3     "$" MUST REFER TO A LABEL OF A FORMAT STATEMENT.

 6 3     AN ENTRY STATEMENT MUST BE LABELED WITH ONE UNSUBSCRIPTED LABEL.

 7 3     A DIMENSION ATTRIBUTE SPECIFIED FOR "$" DOES NOT
         CONTAIN A PARENTHESIZED LIST OF ARRAY BOUNDS.

 8 3     A PROCEDURE STATEMENT MUST BE LABELED WITH ONE UNSUBSCRIPTED LABEL.

 9 3     "$" IS DECLARED WITH AN INVALID RETURNS ATTRIBUTE.

10 2     A DECLARE STATEMENT CANNOT BE LABELED. THE LABEL HAS BEEN IGNORED.

11 3     AN ENTRY STATEMENT CANNOT BE AN IMMEDIATE COMPONENT OF
         A BEGIN-BLOCK, A DO-WHILE STATEMENT, OR AN ITERATIVE DO STATEMENT.

12 3     A PROCEDURE CANNOT BE BOTH A SUBROUTINE AND A FUNCTION.

13 3     INVALID SYNTAX IN THE PARAMETER LIST OF AN ENTRY OR PROCEDURE
         STATEMENT.

14 3     "$" IS A PARAMETER BUT HAS BEEN DECLARED WITH A
         STORAGE CLASS OR USED AS A LABEL.

16 3     INVALID SYNTAX IN AN ENTRY OR PROCEDURE STATEMENT.

17 3     "$" IS AN APPARENT STRUCTURE MEMBER, BUT DOES NOT IMMEDIATELY
         FOLLOW A VARIABLE WITH A LEVEL NUMBER.

18 4     THE DEPTH OF NESTING OF FACTORED DECLARATIONS AND PARAMETER
         OR RETURNS DESCRIPTORS EXCEEDS THE IMPLEMENTATION'S LIMIT
         OF 256.

20 3     INVALID SYNTAX IN A LABEL PREFIX.  THE PREFIX HAS BEEN IGNORED.
```

```
21 3    THIS STATEMENT CONTAINS A DUPLICATE DECLARATION OF "$".

22 4    THE TOTAL NUMBER OF DECLARATIONS, PARAMETER DESCRIPTORS, AND
        RETURNS DESCRIPTORS IN THIS DECLARE STATEMENT EXCEEDS
        THE IMPLEMENTATION'S LIMIT OF 256.

23 3    INVALID SYNTAX IN THE LENGTH SPECIFIED FOR "$".

24 3    INVALID SYNTAX IN DECLARE STATEMENT.

25 3    ATTRIBUTES DECLARED FOR "$" CONFLICT WITH FACTORED
        ATTRIBUTES.

26 3    THIS STATEMENT CONTAINS UNBALANCED PARENTHESES.

27 3    INVALID SYNTAX IN AN APPARENT DO-STATEMENT.

28 3    A FORMAT STATEMENT EXCEEDS THE COMPILER'S LIMIT OF 64
        FORMAT ITEMS INCLUDING THE ENCLOSING PARENTHESES OF THE FORMAT
        LIST. REDUCE THE SIZE OF THE FORMAT BY USING R-FORMATS.

29 3    AN IDENTIFIER CONTAINS MORE THAN 32 CHARACTERS.  ONLY THE FIRST
        32 CHARACTERS WILL BE USED.

30 3    INCLUDED TEXT ENDS WITH AN UNRECOGNIZABLE STATEMENT.

31 3    A FACTORED LEVEL CANNOT BE APPLIED TO "$".

33 2    THE UNDECLARED NAME "$" HAS BEEN DECLARED AS A
        FIXED BINARY(15) PARAMETER.

34 2    A RETURNS ATTRIBUTE ON AN ENTRY OR PROCEDURE STATEMENT CANNOT
        CONTAIN AN ENTRY ATTRIBUTE THAT CONTAINS PARAMETER
        DESCRIPTORS.  USE ONLY THE KEYWORD ENTRY IN THAT CONTEXT.

35 3    THE SUBSCRIPT OF A LABEL PREFIX MUST BE AN OPTIONALLY SIGNED
        INTEGER CONSTANT.

36 3    AN UNRECOGNIZABLE ATTRIBUTE HAS BEEN SPECIFIED FOR "$".

37 3    INVALID SYNTAX IN AN ENTRY ATTRIBUTE SPECIFIED FOR "$".

38 3    INVALID SYNTAX IN THE PRECISION SPECIFIED FOR "$".

39 3    INVALID SCALE-FACTOR SPECIFIED FOR "$".  THE SCALE-
        FACTOR MUST BE AN UNSIGNED INTEGER CONSTANT.

40 3    THE DEFINED ATTRIBUTE SPECIFIED FOR "$" IS NOT IMMEDIATELY
        FOLLOWED BY A PARENTHESIZED BASE REFERENCE.

41 3    THIS STATEMENT CONTAINS AN AMBIGUOUS REFERENCE TO "$".

42 3    "$" IS DECLARED WITH AN INVALID INITIAL ATTRIBUTE.

43 3    "$" IS DECLARED WITH AN INVALID PICTURE ATTRIBUTE.

44 4    THE COMPILER'S UNPAGED STORAGE SPACE IS FULL. BREAK THIS PROGRAM
        INTO SEVERAL EXTERNAL PROCEDURES.  IF YOU COMPILED WITH THE
        OPT OR SUB OPTIONS, TRY COMPILING WITHOUT THEM.

45 3    "$" IS DECLARED WITH CONFLICTING OR DUPLICATE ATTRIBUTES.

46 2    "$" IS DECLARED WITH AN INITIAL ATTRIBUTE BUT IS NOT
        A STATIC VARIABLE.  ONLY STATIC VARIABLES CAN HAVE THE INITIAL
        ATTRIBUTE.
```

```
47 2    "$" IS DECLARED WITHOUT A DATA TYPE.  IT HAS BEEN GIVEN A
        DATA TYPE OF FIXED BINARY(15).

48 3    A RETURNS ATTRIBUTE DESCRIBES AN ARRAY OR A STRUCTURE.
        ALL FUNCTIONS MUST RETURN SCALAR VALUES.

49 2    "$" IS DECLARED AS SCALED FIXED BINARY.  THE SCALE FACTOR HAS
        BEEN IGNORED.  ALL FIXED BINARY VALUES ARE INTEGERS.

50 2    THE PRECISION SPECIFIED FOR "$" EXCEEDS THE IMPLEMENTATION'S
        LIMIT OF FIXED BINARY(15).  THE MAXIMUM PRECISION OF 15 HAS
        BEEN SUPPLIED.  USE FIXED DECIMAL FOR LARGER VALUES.

51 2    THE PRECISION SPECIFIED FOR "$" EXCEEDS THE IMPLEMENTATION'S
        LIMIT OF FIXED DECIMAL(16).  THE MAXIMUM PRECISION OF 16 HAS
        BEEN SUPPLIED.

52 2    THE SCALE FACTOR SPECIFIED FOR "$" EXCEEDS THE SPECIFIED
        PRECISION.  THE SCALE FACTOR HAS BEEN SET TO ZERO.  USE FLOAT
        BINARY FOR VALUES WITH EXCESSIVE SCALE.

53 2    THE PRECISION SPECIFIED FOR "$" EXCEEDS THE IMPLEMENTATION'S
        LIMIT OF FLOAT BINARY(53).  THE MAXIMUM PRECISION OF 53 HAS
        BEEN SUPPLIED.

54 3    COMPILER ERROR: INVALID ENTRY OR LABEL VALUE PASSED TO LOAD_DISPLAY.
        CORRECT ALL PROGRAM ERRORS AND RECOMPILE.  IF THIS MESSAGE PERSISTS,
        CONTACT COMPILER MAINTENANCE PERSONNEL.

55 4    THE TOTAL NUMBER OF SYMBOL TABLE PAGES EXCEEDS THE IMPLEMENTATION'S
        LIMIT OF 256. REDUCE THE NUMBER AND SIZE OF NAMES, CONSTANTS,
        EXTENT-EXPRESSIONS, AND ARGUMENT OR RETURNS DESCRIPTORS.

56 3    THE NUMBER OF DIMENSIONS SPECIFIED FOR "$"
        EXCEEDS THE IMPLEMENTATION'S LIMIT OF 8.

57 3    "$" IS DECLARED WITH AN INVALID DIMENSION ATTRIBUTE.

58 3    A DESCRIPTOR IS AN APPARENT STRUCTURE MEMBER, BUT DOES NOT IMMEDIATELY
        FOLLOW A DESCRIPTOR WITH A LEVEL NUMBER.

59 3    "$" IS DECLARED WITH AN * EXTENT BUT IS NOT
        A PARAMETER OR A DESCRIPTOR.

60 3    "$" IS DECLARED WITH NONCONSTANT EXTENTS BUT IS NOT
        AN AUTOMATIC, BASED, OR DEFINED VARIABLE.

61 4    THE SOURCE TEXT ENDS WITH AN UNRECOGNIZABLE STATEMENT.

62 4    THE SOURCE TEXT DOES NOT BEGIN WITH A PROCEDURE STATEMENT.

63 4    THE LENGTH OF A NAME OR CONSTANT EXCEEDS THE IMPLEMENTATION'S
        LIMIT OF 254 CHARACTERS. CHECK TO SEE IF ALL STRING CONSTANTS
        ARE PROPERLY QUOTED WITH " AND THAT ANY CONTANED "S OCCUR IN
        PAIRS.  ALSO CHECK FOR UNBALANCED /*  */.

64 3    "$" IS REFERENCED WITH TOO MANY SUBSCRIPTS
        OR WITH AN ARGUMENT LIST.

65 3    "$" IS DECLARED WITH A REFER OPTION BUT IS NOT A BASED
        VARIABLE.

66 4    THE DEPTH OF NESTING OF A STRUCTURE EXCEEDS THE
        IMPLEMENTATION'S LIMIT OF 16.
```

| | | |
|---|---|---|
| 67 | 4 | A STATEMENT EXCEEDS THE IMPLEMENTATION'S LIMIT OF 1013 TOKENS (IDENTIFIERS, CONSTANTS, OPERATORS, AND PUNCTUATION SYMBOLS). DECLARE STATEMENTS CAN BE 1013 TOKENS LONG, EXECUTABLE STATEMENTS MUST BE LESS THAN 502 TOKENS LONG. |
| 68 | 3 | INVALID SYNTAX IN APPARENT EXPONENT FIELD OF A FLOATING-POINT CONSTANT. THE EXPONENT HAS BEEN IGNORED. |
| 69 | 3 | AN ARITHMETIC CONSTANT MUST BE SEPARATED FROM THE FOLLOWING SYMBOL BY A DELIMITER. A BLANK DELIMITER HAS BEEN SUPPLIED. |
| 70 | 3 | AN IDENTIFIER BEGINS WITH "_". THE "_" HAS BEEN IGNORED. |
| 71 | 4 | THE TOTAL NUMBER OF ERRORS EXCEEDS THE IMPLEMENTATION'S LIMIT OF 100. |
| 72 | 3 | THE NUMBER OF LABELS ON THIS STATEMENT EXCEEDS THE IMPLEMENTATION'S LIMIT OF 7. |
| 73 | 3 | "$" IS AN ENTRY OR FILE CONSTANT AND CONSEQUENTLY CANNOT BE DIMENSIONED. |
| 74 | 3 | A FUNCTION, SUBROUTINE, BUILT-IN FUNCTION, OR PSEUDO-VARIABLE REFERENCE APPEARS IN A CONTEXT THAT REQUIRES A VARIABLE REFERENCE. |
| 75 | 3 | "$" IS DECLARED WITH MIXED * AND CONSTANT BOUNDS. IF ANY BOUND IS *, ALL BOUNDS MUST BE *. |
| 76 | 3 | ONE OF THE BOUNDS DECLARED FOR "$" IS INVALID BECAUSE THE LOWER BOUND IS GREATER THAN THE UPPER BOUND. |
| 77 | 3 | THIS STATEMENT CONTAINS AN UNRECOGNIZABLE CONDITION NAME. |
| 78 | 4 | COMPILER ERROR. PASS2 ATTEMPTED TO READ PAST THE END OF THE TOKEN FILE. CORRECT ALL SOURCE PROGRAM ERRORS AND RECOMPILE. IF THIS MESSAGE PERSISTS, CONTACT COMPILER MAINTENANCE PERSONNEL. |
| 79 | 4 | EXCESSIVE COMPLEXITY OF THIS STATEMENT HAS CAUSED THE COMPILER'S PARSE OUTPUT STACK TO OVERFLOW. REDUCE THE COMPLEXITY AND RECOMPILE. |
| 80 | 4 | THIS STATEMENT CONTAINS A REFERENCE THAT HAS MORE THAN 31 LEVELS OF POINTER QUALIFICATION. |
| 81 | 3 | THIS GET OR PUT STATEMENT CONTAINS AN UNRECOGNIZABLE OPTION. |
| 82 | 3 | SYNTAX ERROR. EXPECTED $ NOT FOUND. |
| 83 | 3 | SYNTAX ERROR. EXPECTED "$" KEYWORD NOT FOUND. |
| 84 | 3 | THIS STATEMENT CONTAINS AN UNRECOGNIZABLE FORMAT ITEM. |
| 85 | 3 | SYNTAX ERROR. AN ELSE-CLAUSE APPEARS WITHOUT A MATCHING THEN-CLAUSE. |
| 86 | 3 | THE VALUE OF AN INTEGER CONSTANT USED IN A FORMAT LIST EXCEEDS THE IMPLEMENTATION'S LIMIT OF 255. |
| 87 | 3 | A STRUCTURE QUALIFIED REFERENCE CONTAINS MORE THAN 15 QUALIFYING NAMES. |
| 88 | 3 | A FUNCTION REFERENCE CONTAINS MORE THAN 16 ARGUMENTS. |

```
89 3     A THEN-CLAUSE OR AN ELSE-CLAUSE CANNOT CONSIST OF:
         A DECLARE STATEMENT, A FORMAT STATEMENT, A PROCEDURE STATEMENT,
         AN ENTRY STATEMENT, OR AN END STATEMENT.

90 3     A CASE CANNOT CONSIST OF: A DECLARE STATEMENT, A FORMAT STATEMENT,
         A PROCEDURE STATEMENT, OR AN ENTRY STATEMENT.

91 3     SYNTAX ERROR. AN EXPECTED IDENTIFIER WAS NOT FOUND FOLLOWING
         THE KEYWORD OF AN ALLOCATE STATEMENT.

92 3     THE UNDECLARED IDENTIFIER "$" CANNOT BE USED IN AN ALLOCATE
         STATEMENT BECAUSE IF IT WERE DECLARED BY THE COMPILER IT
         WOULD HAVE THE AUTOMATIC STORAGE CLASS.

93 3     "$" APPEARS IN AN ALLOCATE STATEMENT, BUT IS NOT A
         LEVEL-ONE BASED VARIABLE.

94 3     THIS OPEN STATEMENT CONTAINS MULTIPLE TITLE-OPTIONS.

95 3     THIS OPEN STATEMENT CONTAINS MULTIPLE PAGESIZE-OPTIONS.

96 3     THIS OPEN STATEMENT CONTAINS MULTIPLE LINESIZE-OPTIONS.

97 3     "$" IS AN UNRECOGNIZABLE FILE DESCRIPTION ATTRIBUTE.

98 3     A FUNCTION, SUBROUTINE, OR BUILT-IN FUNCTION REFERENCE
         APPEARS ON THE LEFT-SIDE OF AN ASSIGNMENT STATEMENT.

99 3     THIS STATEMENT CONTAINS INCONSISTENT FILE DESCRIPTION ATTRIBUTES.

100 3    THE UNDECLARED IDENTIFIER "$" APPEARS IN A CONTEXT THAT
         IS EITHER A FUNCTION REFERENCE, A CALL, OR AN ARRAY REFERENCE.
         IT HAS NOT BEEN DECLARED BY THE COMPILER.

101 2    THE UNDECLARED IDENTIFIER "$" HAS BEEN DECLARED
         AS A FIXED BINARY INTEGER IN THE IMMEDIATELY CONTAINING BLOCK.

102 3    THE ARGUMENT TO THE SIZE BUILT-IN FUNCTION IS NOT AN IDENTIFIER.

103 3    THE UNDECLARED IDENTIFIER "$" APPEARS AS AN ARGUMENT TO THE
         SIZE BUILT-IN FUNCTION.  IT HAS NOT BEEN DECLARED BY THE
         COMPILER.

104 3    "$" IS NOT A LEVEL-ONE VARIABLE.  THE ARGUMENT TO THE SIZE
         BUILT-IN FUNCTION MUST BE A LEVEL-ONE  VARIABLE.

105 3    AN END STATEMENT APPEARS WITHOUT A MATCHING: DO, BEGIN,
         OR PROCEDURE STATEMENT.

106 3    "$" HAS BEEN REFERENCED WITH TOO MANY SUBSCRIPTS.

107 3    "$" HAS BEEN REFERENCED WITH FEWER SUBSCRIPTS THAN
         IT HAS DIMENSIONS.

108 3    "$" HAS BEEN DECLARED AS A BUILT-IN FUNCTION, BUT
         IS NOT KNOWN TO THIS IMPLEMENTATION.

109 3    A BUILT-IN FUNCTION REFERENCE APPEARS IN A CONTEXT THAT
         REQUIRES A SUBROUTINE REFERENCE.

110 3    A BIT-STRING CONSTANT WRITTEN IN B2, B3, OR B4 FORM EXCEEDS
         THE IMPLEMENTATION'S LIMIT OF 255 BITS WHEN EXPANDED.
```

```
111 4   THE ARGUMENT DESCRIPTOR REQUIRED BY A CALL FROM THIS STATEMENT
        EXCEEDS THE IMPLEMENTATION'S LIMIT OF 127 WORDS.  DO NOT PASS
        EXCESSIVELY LARGE STRUCTURES TO ENTRY POINTS THAT REQUIRE
        DESCRIPTORS.

112 4   "$" HAS AN EXTENT EXPRESSION THAT REQUIRES THE CREATION OF
        AN ARGUMENT DESCRIPTOR.  IT CANNOT BE PASSED TO AN ENTRY POINT
        THAT REQUIRES DESCRIPTORS.

113 3   A BIT-STRING CONSTANT CONTAINS AN INVALID DIGIT FOLLOWING
        THE B.  ONLY 1, 2, 3, OR 4 ARE PERMITTED.

114 3   THIS IS AN UNRECOGNIZABLE % STATEMENT.

115 3   INVALID SYNTAX IN %REPLACE STATEMENT. THE CORRECT SYNTAX
        IS %REPLACE ID BY C; WHERE ID IS AN IDENTIFIER AND C IS: A
        CHARACTER-STRING CONSTANT, A BIT-STRING CONSTANT, OR AN
        ARITHMETIC CONSTANT PRECEDED BY AN OPTIONAL MINUS SIGN.

116 3   COMPILER ERROR IN CONVERSION OF A CONSTANT TO ITS INTERNAL
        REPRESENTATION.  CORRECT ALL PROGRAM ERRORS AND RECOMPILE.  IF
        THIS MESSAGE PERSISTS, CONTACT COMPILER MAINTENANCE PERSONNEL.

117 4   NESTED %INCLUDE STATEMENTS CANNOT BE NESTED MORE
        THAN 4 LEVELS DEEP.

118 3   INVALID SYNTAX IN %INCLUDE STATEMENT.  THE CORRECT SYNTAX
        IS %INCLUDE <CHAR-STRING-CONSTANT>;

119 3   THE FIRST ARGUMENT TO A LBOUND, HBOUND, OR DIM BUILT-IN
        FUNCTION MUST BE AN ARRAY REFERENCE.

120 3   "$" HAS FEWER DIMENSIONS THAN ARE REQUIRED BY THE BUILT-IN
        FUNCTION REFERENCE, OR THE FUNCTION REFERENCE HAS A ZERO AS
        ITS SECOND ARGUMENT.

121 2   THE INITIAL ATTRIBUTE SPECIFIED FOR "$"
        CONTAINS FEWER VALUES THAN ARE REQUIRED TO FULLY INITIALIZE
        THE VARIABLE.

122 3   "$" IS A BASED VARIABLE REFERENCED WITHOUT A POINTER.

123 3   "$" IS A NONBASED VARIABLE REFERENCED WITH A POINTER.

124 3   ONE OF THE INITIAL VALUES SPECIFIED FOR "$"
        CANNOT BE CONVERTED TO THE TYPE OF THE VARIABLE.

125 3   "$" CANNOT BE PROPERLY INITIALIZED BECAUSE IT
        IS AN UNCONNECTED ARRAY THAT SHARES ONE OR MORE WORDS WITH
        ANOTHER UNCONNECTED AND INITIALIZED ARRAY.  REWRITE THE
        DECLARATION SO THAT THE ARRAYS ARE NOT UNCONNECTED.

126 2   EXCESS INITIAL VALUES HAVE BEEN SPECIFIED FOR "$".

127 3   $ IS AN INVALID PICTURE.

128 3   $ SPECIFIES MORE THAN 16 DECIMAL DIGITS.

129 3   $
        RESULTS IN A COMPILED PICTURE THAT EXCEEDS THE
        IMPLEMENTATION'S LIMIT. REDUCE THE SIZE OF THE PICTURE
        AND RECOMPILE.

130 3   $ CONTAINS MULTIPLE DOLLAR SYMBOLS.
```

```
131 3   $ CONTAINS MULTIPLE SIGN SYMBOLS.

132 3   A RETURNS ATTRIBUTE SPECIFIES AN ASTERISK LENGTH.
        ALL LENGTHS SPECIFIED IN RETURNS ATTRIBUTES MUST BE CONSTANTS.

133 1   WARNING: ASSIGNMENTS OF THE FORM X = P; WHERE P IS PICTURED
        AND X IS CHAR OR PICTURED, ARE COMPILED AS X = FIXED(P); STANDARD
        PL/I WOULD DO A CHARACTER STRING ASSIGNMENT.  USE
        DECLARE C CHAR(N) DEFINED(P); TO REFER TO P AS A CHARACTER STRING.

134 2   STANDARD PL/I DOES NOT DEFINE CONVERSON BETWEEN INTEGER
        AND POINTER DATA.  THE USE OF INTEGERS AS POINTERS CAN PRODUCE INVALID
        PROGRAMS WHEN THE OPTIMIZE OPTION IS SPECIFIED.

135 3   NONPOINTER DATA APPEARS IN A CONTEXT THAT EXPECTS POINTER DATA.
        NO CONVERSION IS PERFORMED.

136 3   X = X!!Y; IS MEANINGFUL ONLY WHEN X IS A CHARACTER VARYING VARIABLE.

137 1   WARNING: FIXED BINARY DIVISION BY THE / OPERATOR PRODUCES A FIXED
        BINARY RESULT.  STANDARD PL/I PRODUCES A SCALED BINARY RESULT WITH
        A MAXIMAL BINARY FRACTION.  USE DIVIDE(X,Y,15) TO CONFORM TO STANDARD
        PL/I.

138 1   WARNING: CEIL, FLOOR, OR TRUNC OF A FIXED BINARY VALUE HAS NO EFFECT.

139 1   WARNING: CEIL, FLOOR, OR TRUNC OF A FIXED DECIMAL(P,0) VALUE HAS
        NO EFFECT.

140 3   THE ARGUMENT OF ADDR CANNOT BE BIT ALIGNED.

141 1   WARNING: FLOAT(X) WHERE X IS DECIMAL OR CHARACTER, PRODUCES A FLOAT
        BINARY RESULT.  STANDARD PL/I WOULD PRODUCE A FLOAT DECIMAL RESULT.
        THE RESULT VALUE IS THE SAME IN EITHER CASE, BUT THE PRECISION IS
        MEASURED IN DIFFERENT UNITS IN THE TWO CASES.

142 1   WARNING: BINARY(X) WHERE X IS FIXED DECIMAL(P,Q), PRODUCES A FIXED
        BINARY INTEGER.  STANDARD PL/I WOULD PRODUCE A SCALED BINARY RESULT.
        WRITE BINARY(X,15) TO CONFORM TO STANDARD PL/I.

143 1   WARNING: DECIMAL(X) WHERE X IS FLOAT BINARY, PRODUCES A
        FIXED DECIMAL RESULT.  STANDARD PL/I WOULD PRODUCE A FLOAT DECIMAL
        RESULT.  WRITE DECIMAL(FIXED(X,15)) TO CONFORM TO STANDARD PL/I.

144 3   ARITHMETIC OPERATORS REQUIRE ARITHMETIC OPERANDS.  ALSO,
        SUBSCRIPTS MUST BE FIXED BINARY EXPRESSIONS.  WRITE AN EXPLICIT
        CONVERSION FUNCTION (DECIMAL,FIXED,FLOAT, OR BINARY) TO CONVERT
        A STRING OPERAND TO ARITHMETIC TYPE.

145 3   THE OPERANDS OF A RELATIONAL OPERATOR MUST BOTH BE ARITHMETIC,
        MUST BOTH BE CHARACTER STRINGS, MUST BOTH BE BIT STRINGS, OR MUST
        BOTH BE THE SAME DATA TYPE.

146 1   WARNING: ASSIGNMENT OF AN ARITHMETIC VALUE TO A BIT-STRING IS LIKELY
        TO PRODUCE AN UNEXPECTED RESULT BECAUSE THE CONVERTED
        VALUE IS ASSIGNED FROM LEFT-TO-RIGHT WITH TRUNCATION OF EXCESS BITS.
        WRITE B = SUBSTR(FIXED(A),I,J) TO ASSIGN THE RIGHTMOST J BITS.

147 2   CONVERSION FROM POINTER TO FIXED BINARY IS NOT DEFINED
        BY STANDARD PL/I.

148 3   COMPILER ERROR: THE WORD_SIZE ROUTINE HAS BEEN GIVEN A
        VARIABLE SIZE VALUE. CORRECT ALL PROGRAM ERRORS AND RECOMPILE.  IF THIS
        MESSAGE PERSISTS, CONTACT COMPILER MAINTENANCE PERSONNEL.
```

```
149 3   COMPILER ERROR: THE FIND_VALUE ROUTINE COULD NOT FIND A VALUE.  CORRECT
        ALL PROGRAM ERRORS AND RECOMPILE.  IF THIS MESSAGE PERSISTS, CONTACT
        COMPILER MAINTENANCE PERSONNEL.

150 3   COMPILER ERROR: THE LOAD_REGISTER ROUTINE HAS BEEN ASKED TO LOAD
        A LOCKED REGISTER.  CORRECT ALL PROGRAM ERRORS AND RECOMPILE.  IF THIS
        MESSAGE PERSISTS, CONTACT COMPILER MAINTENANCE PERSONNEL.

151 3   COMPILER ERROR: THE SELECT_REGISTER ROUTINE HAS BEEN UNABLE TO SELECT
        AN UNLOCKED REGISTER.  CORRECT ALL PROGRAM ERRORS AND RECOMPILE.
        IF THIS MESSAGE PERSISTS, CONTACT COMPILER MAINTENANCE PERSONNEL.

152 3   COMPILER ERROR: THE LOAD_REGISTER ROUTINE HAS BEEN ASKED TO LOAD AN AC
        WITH A VALUE THAT IS IN FAC.  CORRECT ALL PROGRAM ERRORS AND RECOMPILE.
        IF THIS MESSAGE PERSISTS, CONTACT COMPILER MAINTENANCE PERSONNEL.

153 3   COMPILER ERROR: THE LOAD_REGISTER ROUTINE HAS BEEN ASKED TO LOAD A FAC
        WITH A VALUE THAT IS IN AN AC OR THAT IS IMMEDIATE.  CORRECT ALL PROGRAM
        ERRORS AND RECOMPILE.  IF THIS MESSAGE PERSISTS, CONTACT COMPILER
        MAINTENANCE PERSONNEL.

154 1   WARNING: CHAR(P) WHERE P IS PICTURED, IS COMPILED AS CHAR(FIXED(P)).
        STANDARD PL/I WOULD PRODUCE THE CHARACTER STRING VALUE OF P.
        USE DECLARE C CHAR(N) DEFINED(P); TO REFER TO P AS CHARACTER STRING.

155 4   A NESTED BLOCK REFERS TO A VARIABLE DECLARED
        MORE THAN 16 BLOCK LEVELS OUT.  THIS EXCEEDS THE CODE GENERATOR'S
        DISPLAY STACK.

156 3   "$" CANNOT BE PASSED BY-REFERENCE BECAUSE IT DOES
        NOT MATCH THE CORRESPONDING PARAMETER.  IT CANNOT BE PASSED BY-VALUE
        BECAUSE EITHER IT OR THE PARAMETER IS AN ARRAY OR STRUCTURE.

157 3   AN ARRAY OR STRUCTURE HAS BEEN REFERENCED IN
        A CONTEXT THAT REQUIRES A SCALAR VALUE.

158 3   THE ARGUMENT TO THE VALID BUILT-IN FUNCTION MUST BE
        A PICTURED VARIABLE OR PICTURE VALUED FUNCTION.

159 3   COMPILER ERROR: ATTEMPT TO FREE AN ALREADY FREED VALUE NODE.
        CORRECT ALL PROGRAM ERRORS AND RECOMPILE.  IF THIS MESSAGE PERSISTS,
        CONTACT COMPILER MAINTENANCE PERSONNEL.

160 3   "$" IS AN ARRAY OR STRUCTURE THAT CANNOT BE PASSED AS AN ARGUMENT
        BECAUSE IT DOES NOT MATCH THE PARAMETER TO WHICH IT IS PASSED.

161 3   INVALID OPTION IN DO STATEMENT.  THE VALID FORMS OF THE
        DO ARE: DO; DO WHILE (<EXP>); DO <REF> = <EXP> TO <EXP> [BY <EXP>];
        DO <REF> = <EXP> [,<EXP>...]; DO CASE (<EXP>);

162 1   WARNING: MIXED DECIMAL AND BINARY OPERANDS ARE CONVERTED TO DECIMAL.
        STANDARD PL/I WOULD CONVERT THEM TO BINARY.  WRITE THE DECIMAL
        BUILT-IN FUNCTION AROUND THE BINARY OPERAND TO CONFORM TO STANDARD
        PL/I.

163 3   A BUILT-IN FUNCTION HAS BEEN GIVEN THE WRONG NUMBER OF ARGUMENTS.

164 3   THE DATA TYPE OF A VALUE USED IN THIS STATEMENT IS INCORRECT FOR THE
        CONTEXT IN WHICH THE VALUE APPEARS.

165 3   A MULTIPLY OPERATOR IN THIS STATEMENT PRODUCES A VALUE WHOSE SCALE-
        FACTOR EXCEEDS THE IMPLEMENTATION'S LIMIT OF 16.  A SCALE-FACTOR
        OF 16 HAS BEEN USED.
```

```
166 3   A DIVIDE OPERATOR IN THIS STATEMENT PRODUCES A VALUE WHOSE SCALE-FACTOR
        IS OUTSIDE OF THE RANGE 0<=G<=16.  A SCALE-FACTOR OF 0 HAS BEEN USED.

167 3   A SUBROUTINE HAS BEEN INVOKED AS A FUNCTION.

168 3   THE ARGUMENT OF THE STRING BUILT-IN FUNCTION OR PSEUDO-VARIABLE MUST BE
        AN ARRAY OR STRUCTURE.

169 1   WARNING: A FIXED DECIMAL VALUE RAISED TO AN INTEGER POWER PRODUCES A
        FLOAT BINARY RESULT.  STANDARD PL/I WOULD PRODUCE A FLOAT DECIMAL
        RESULT IN THIS CASE.  WRITE BINARY(X)**C TO CONFORM TO STANDARD PL/I.

170 3   A BUILT-IN FUNCTION REFERENCE HAS THE WRONG NUMBER OF ARGUMENTS.

171 3   ONE OF THE ARGUMENTS THAT SPECIFIES THE RESULT PRECISION
        OF A BUILT-IN FUNCTION IS OUTSIDE OF THE RANGE PERMITTED BY
        THE IMPLEMENTATION.  THE MAXIMUM PRECISIONS ARE: FIXED BINARY(15),
        FIXED DECIMAL(16,16), AND FLOAT BINARY(53).

172 3   A MEMBER OF A DIMENSIONED STRUCTURE IS AN ARRAY WHOSE STORAGE
        IS UNCONNECTED.  AS SUCH IT CANNOT APPEAR AS THE ARGUMENT
        TO UNSPEC, STRING OR ADDR.

173 4   COMPILER ERROR: PASS3 DESTROYED SOME OF ITS VARIABLES
        WHEN IT RELOADED STORAGE.  CONTACT COMPILER MAINTENANCE
        PERSONNEL.

174 3   A BIT ALIGNED VARIABLE, X , CANNOT APPEAR IN AN INTO(X) OR
        FROM(X) CONTEXT.

175 4   COMPILER ERROR: THE CODE GENERATOR'S STACK HAS OVERFLOWED.
        CORRECT ALL PROGRAM ERRORS AND RECOMPILE.  IF THIS MESSAGE PERSISTS,
        CONTACT COMPILER MAINTENANCE PERSONNEL.

176 3   INSUFFICIENT ARGUMENTS IN A CALL TO "$".

177 3   TOO MANY ARGUMENTS IN A CALL TO "$".

178 1   WARNING: "$" HAS BEEN PASSED BY-VALUE BECAUSE ITS
        ATTRIBUTES DID NOT MATCH THOSE OF THE CORRESPONDING PARAMETER.
        ENCLOSE THE ARGUMENT IN PARENTHESES TO SUPPRESS THIS WARNING.

179 3   THE RIGHT SIDE OF AN ASSIGNMENT TO AN ARRAY OR STRUCTURE MUST BE
        AN ARRAY OR STRUCTURE THAT IS IDENTICAL IN SHAPE, SIZE, AND
        COMPONENT DATA TYPES WITH THE LEFT SIDE.  NEITHER THE LEFT OR
        RIGHT SIDE CAN BE A MEMBER OF A DIMENSIONED STRUCTURE.


180 4   AN EXECUTABLE STATEMENT EXCEEDS THE IMPLEMENTATION'S LIMIT
        OF 501 TOKENS.  REDUCE THE LENGTH OF THE STATEMENT.

181 4   COMPILER ERROR: AN INVALID SYMBOL PAGE NUMBER HAS BEEN
        PASSED TO THE FIND_PAGE ROUTINE. CORRECT ALL ERRORS AND RECOMPILE.
        IF THIS MESSAGE PERSISTS, CONTACT COMPILER MAINTENANCE PERSONNEL.

182 3   THE SECOND ARGUMENT OF THE ROUND BUILT-IN FUNCTION MUST BE AN
        INTEGER CONSTANT, K, THAT SATISFIES 0<=K<=Q WHEN THE FIRST ARGUMENT
        IS DECIMAL(P,Q), OR THAT SATISFIES 1<=K<=53 WHEN THE FIRST ARGUMENT
        IS FLOAT.

183 1   WARNING: RETURN STATEMENTS IN ON-UNITS RETURN TO THE CALLER OF THE
        PROCEDURE THAT IMMEDIATELY CONTAINS THE ON-UNIT.  THEY DO NOT RETURN
        TO THE SOURCE OF THE SIGNAL.
```

```
184 3   RETURN(EXP) IS NOT VALID IN A SUBROUTINE PROCEDURE.

185 3   RETURN; IS NOT VALID IN A FUNCTION PROCEDURE.

186 3   MIN AND MAX ARE RESTRICTED TO TWO ARGUMENTS.
        USE MIN(A,MIN(B,C)) ETC. FOR MULTIPLE ARGUMENTS.

187 3   INVALID OPTION OR WRONG ORDER OF OPTIONS IN A RECORD I/O
        STATEMENT.  THE CORRECT ORDER IS: FILE(), INTO() OR FROM(),
        KEY() OR KEYFROM().

188 3   "$" APPEARS IN A CONTEXT THAT REQUIRES AN ENTRY VALUE.

189 3   "$" HAS BEEN DECLARED WITH A RETURNS ATTRIBUTE, BUT DOES NOT
        HAVE THE ENTRY ATTRIBUTE.  AN EMPTY ENTRY ATTRIBUTE HAS BEEN
        SUPPLIED.

190 2   TEXT FOLLOWS THE KEYWORD "END", IT HAS BEEN IGNORED.

192 3   A SUBSCRIPT, STRING LENGTH, OR ARRAY BOUND MUST BE A FIXED
        BINARY VALUE.

193 3   A NONPOINTER VALUE HAS BEEN USED AS A POINTER QUALIFIER.

197 4   THE NESTING OF DO, PROCEDURE, AND BEGIN STATEMENTS
        EXCEEDS THE IMPLEMENTATION'S LIMIT OF 31.

198 4   COMPILER ERROR: INVALID NODE ID PASSED TO THE
        FIND_NODE ROUTINE.  CORRECT ALL PROGRAM ERRORS AND RECOMPILE.
        IF THIS MESSAGE PERSISTS, CONTACT COMPILER MAINTENANCE
        PERSONNEL.  IF ALL ELSE FAILS, COMPILE WITHOUT THE D OPTION.

200 4   THE NUMBER OF %INCLUDE STATEMENTS EXCEEDS THE IMPLEMENTATION'S
        LIMIT OF 32.

205 3   AN ENTRY NAME HAS BEEN USED IN A CONTEXT THAT REQUIRES
        SOME OTHER DATA TYPE.  FUNCTIONS THAT HAVE NO ARGUMENTS
        MUST BE INVOKED BY USE OF AN EMPTY () ARGUMENT LIST.

207 3   "$" IS A FUNCTION AND CANNOT BE CALLED AS A SUBROUTINE.

208 3   THIS ARGUMENT OF UNSPEC CANNOT BE ADDRESSED AS A BIT STRING
        BECAUSE IT IS BYTE ALIGNED.

209 3   SYNTAX ERROR.  AN OTHERWISE-CLAUSE APPEARS WITHOUT A
        MATCHING DO CASE STATEMENT.

210 2   COMMENTS MAY NOT CONTAIN THE CHARACTER SEQUENCE /* .
```

End of Appendix

# Appendix E
# Runtime Error Messages

The default on-unit for the ERROR condition will write one of the following messages to @OUTPUT before terminating execution of your program. The number preceding each message is the value returned by the ONCODE built-in function for each error.

| | |
|---|---|
| 1 | ERROR CONDITION SIGNALLED BY A SIGNAL STATEMENT. |
| 2 | SYSTEM ERROR READING A STREAM FILE. |
| 3 | ATTEMPT TO READ A DIRECT FILE WITHOUT SUPPYLING A KEY. |
| 4 | SYSTEM ERROR READING A KEYED FILE. |
| 5 | ATTEMPT TO READ A NON-KEYED FILE WITH A KEY OPTION. |
| 6 | SYSTEM ERROR READING A NON-KEYED FILE. |
| 7 | UNABLE TO OBTAIN A SYSTEM I/O CHANNEL. |
| 8 | SYSTEM ERROR ON ATTEMPTED OPEN. |
| 9 | LINESIZE EXCEEDS IMPLEMENTATION'S LIMIT OF 132. |
| 10 | SYSTEM ERROR ON ATTEMPT TO CREATE A RANDOM FILE. |
| 11 | SYSTEM ERROR ON ATTEMPT TO CREATE A SEQUENTIAL FILE. |
| 12 | SYSTEM ERROR ON ATTEMPT TO CLOSE A FILE. |
| 13 | ATTEMPT TO WRITE ON AN INPUT FILE. |
| 14 | SYSTEM ERROR WRITING TO A STREAM FILE. |
| 15 | ATTEMPT TO WRITE A DIRECT FILE WITHOUT SUPPLYING A KEY. |
| 16 | SYSTEM ERROR WRITING A KEYED FILE. |
| 17 | ATTEMPT TO WRITE ON A NON-KEYED FILE WITH A KEY. |
| 18 | SYSTEM ERROR WRITING A NON-KEYED FILE. |
| 19 | SYSTEM ERROR ON ATTEMPT TO DELETE AN EXISTING FILE. |
| 20 | ATTEMPT TO WRITE MORE THAN 132 CHARACTERS TO A STREAM FILE. |
| 21 | ATTEMPT TO READ MORE THAN 256 WORDS FROM A RECORD FILE. |
| 22 | ATTEMPT TO WRITE MORE THAN 256 WORDS TO A RECORD FILE. |
| 23 | ATTEMPT TO REWRITE TO A FILE THAT IS NOT KEYED UPDATE. |
| 24 | ATTEMPT TO DELETE A RECORD FROM A FILE THAT IS NOT KEYED UPDATE. |
| 25 | STACK AND AREA STORAGE IS FULL. |
| 26 | ATTEMPT TO PUT TO A FILE THAT IS NOT STREAM OUTPUT. |
| 27 | RECURSIVE USE OF I/O STATEMENTS, R-FORMATS, OR NESTED FORMAT LISTS HAS CAUSED I/O STORAGE TO OVERFLOW. |

| | |
|---|---|
| 28 | A CHARACTER STRING CONTAINS INVALID DATA FOR CONVERSION TO ARITHMETIC OR BIT TYPE. |
| 29 | FIXED OVERFLOW DURING TYPE CONVERSION. |
| 30 | A SUBSCRIPT, SUBSTR, OR CASE INDEX IS OUT OF RANGE. |
| 31 | ARITHMETIC OVERFLOW DURING TYPE CONVERSION. |
| 32 | ATTEMPT TO GET FROM A FILE THAT IS NOT STREAM INPUT. |
| 33 | THE DATA TYPE OF AN OUTPUT VALUE IS INVALID FOR USE WITH THE CORRESPONDING FORMAT ITEM. |
| 34 | THE DATA TYPE OF AN INPUT VARIABLE IS INVALID FOR USE WITH THE CORRESPONDING FORMAT ITEM. |
| 35 | ATTEMPT TO PUT-LIST DATA THAT IS OTHER THAN STRING OR ARITHMETIC. |
| 36 | ATTEMPT TO GET-LIST DATA THAT IS OTHER THAN STRING OR ARITHMETIC. |
| 37 | MISSING FIELD SIZE IN A,B,B1,B2,B3 OR B4 FORMAT ITEMS. |
| 38 | ATTEMPT TO USE A PAGE, LINE, OR TAB FORMAT ITEM ON A FILE THAT DOES NOT HAVE THE PRINT ATTRIBUTE. |
| 39 | AN INVALID CHARACTER FOLLOWS A QUOTED STRING IN A STREAM INPUT FILE BEING PROCESSED BY A GET-LIST. |
| 40 | ATTEMPT TO OPEN A NON-EXISTENT FILE FOR INPUT. |
| 41 | INSUFFIENT FIELD WIDTH FOR AN E FORMAT ITEM. |
| 42 | INSUFFIENT FIELD WIDTH FOR B FORMAT ITEM. |
| 43 | ATTEMPT TO DELETE A RECORD WITHOUT SUPPLYING A KEY. |
| 44 | COMMERCIAL INSTRUCTION FAULT. |
| 45 | INVALID DATA TYPE FOR CONVERSION. |
| 46 | OVERFLOW ON EXPONENTIATION. |
| 47 | INVALID ARGUMENT FOR ASIN OR ACOS. |
| 48 | NEGATIVE OR ZERO ARGUMENT FOR LOG, LOG2 OR LOG10. |
| 49 | REQUEST FOR ATAN AT ORIGIN. |
| 50 | NEGATIVE ARGUMENT FOR SQRT. |
| 51 | INVALID ARGUMENT FOR EXPONENTIATION. |
| 52 | ATTEMPT TO READ AN OUTPUT FILE. |
| 53 | SYSTEM ERROR ON ATTEMPT TO CREATE A FILE. |
| 54 | WRONG HARDWARE CONFIGURATION FOR PROGRAM. |
| 55 | BIND OF PROGRAM FAILED TO SPECIFY START LOCATION (NAME/F). |
| 56 | FLOATING POINT OVERFLOW. |
| 57 | FLOATING POINT ATTEMPT TO DIVIDE BY ZERO. |

End of Appendix

# Appendix F
# The SYS Library Routine: AOS System Calls from PL/I

The SYS function is provided as part of the AOS PL/I library to enable AOS system calls to be made from PL/I programs. The function takes four arguments. The first argument is a BIT(16) ALIGNED string containing the code of the AOS system call to be made. The other three arguments specify the values accumulators 0, 1, and 2 are to have when the system call is made.

To use SYS, it must be declared in the PL/I program as an external entry constant with four arguments. The first argument must be BIT(16) ALIGNED. The others may be BIT(16) ALIGNED, POINTER, or FIXED BINARY(15), depending on which is most convenient for the system calls to be made. It may be necessary, using DEFINED, for variables with these three data types to share words of storage to pass arguments to SYS efficiently.

Many AOS system calls require a packet, or several words of information which the system needs to carry out the system call. Packets must be set up by the user using an array or structure. The address of the packet is passed as one of the arguments to SYS.

To make SYS easier to use, the files PL1SYSID.PL1 and PL1PARU.PL1 are provided with the compiler. PL1SYSID.PL1 contains %REPLACE declarations for the system call codes. For example, the code for the ?DELETE system call is "0001"B4. PL1SYSID.PL1 contains the following line:

```
%REPLACE SYS_DELETE BY "0001"B4;
```

This enables the identifier SYS_DELETE to be used as the first argument to SYS. The file PL1PARU.PL1 contains %REPLACE declarations for parameter offsets within packets.

For details of AOS system calls, see the AOS Programmer's Manual (093-000120).

Many of the system calls return information in accumulators 0, 1, and 2. In order to access this information, you must be sure that the second, third, and fourth arguments to SYS are passed by reference. They must be FIXED BINARY(15), POINTER, or BIT(16) ALIGNED variables. After the call to SYS, these variables will contain the contents of accumulators 0, 1, and 2 returned by the system call.

The SYS function call returns the value "0"B if the system call was performed successfully. If an AOS error occurred, it returns the value "1"B. If an error occurred, the AOS exceptional condition code is returned through AC0 to the second argument of SYS.

Example:

```
GET_SWITCH:
        PROCEDURE(SW) RETURNS(BIT);

DECLARE SW CHARACTER(*);
DECLARE SWITCH CHARACTER(33);
DECLARE 1 PACKET,
        2 TYPE FIXED,
        2 ARG FIXED,
        2 SW_PTR POINTER,
        2 KEY_PTR POINTER;
DECLARE SW_LEN FIXED
DECLARE KEYWORD CHARACTER(33);
DECLAE X FIXED;
DECLARE SYS ENTRY(BIT(16) ALIGNED, FIXED BINARY, FIXED BINARY, POINTER)
                                RETURNS(BIT);

SWITCH=SW!!ASCII(0);                    /* APPEND NUL */
TYPE=SYS_GTSW;                          /* DEFINED IN PL1PARU.PL1 */
ARG=0;                                  /* WANT GLOBAL SWITCH */
SW_PTR=ADDR(SWITCH);                    /* BYTE POINTER TO SWITCH */
KEY_PTR=ADDR(KEYWORD);                  /* BYTE POINTER TO KEYWORD */

IF SYS(SYS_GTMES, LENGTH, X ADDR(PACKET)) THEN;

IF LENGTH=-1 THEN RETURN ("0"B);        /* SWITCH DOESN'T EXIST */
RETURN ("1"B);                          /* SIMPLE OR KEYWORD SWITCH */
END;
```

This procedure returns a "1"B if the executing program was invoked with a particular global switch on the command line, else it returns a "0"B. It presumes that PL1SYSID.PL1 and PL1PARU.PLI have been included with %INCLUDE earlier in the program. For details on the ?GTMES command, see the AOS Programmer's Manual.

End of Appendix

# Appendix G
# Calling Assembly Language Procedures

ECLIPSE assembly language procedures may be called from a PL/I program. To the PL/I program, an assembly language procedure looks exactly like an external PL/I procedure. The procedure may be used in a CALL statement or as a function reference. It must be declared as an external entry constant in a DECLARE statement, and the data types of the arguments and of the returned value, if any, must be described.

The assembly language procedure must conform to the calling sequence of PL/I. That is, the procedure must know how to address the argument passed to it and where to store a result value. It must return to the calling program, leaving the stack and the machine registers in the proper state.

The following steps describe the details of the PL/I calling sequence which are necessary to understand in order to write an assembly language procedure.

When a procedure is invoked:

- The address of the argument to the procedure are pushed onto the stack, in reverse order. This leaves the address of the first argument on the top of the stack. If an argument is passed by reference, the address pushed is the actual address of the argument. If it is passed by value, the address pushed is the location of a temporary into which the value of the argument has been copied.
- If it is a function call, the address of where to store the result of the function is pushed onto the stack.
- An EJSR instruction loads the return address into AC3 and jumps to the called procedure.

On return from the procedure:

- The values of AC0, AC1, and AC2 must be unchanged from when before the procedure was called.
- The stack frame pointer must be unchanged. AC3 must also contain the frame pointer.

The above rules can most easily be followed by doing the following in the assembly language procedure:

- Begin with a SAVE instruction. This saves the contents of AC0, AC1, AC2, the frame pointer, and the return address on the stack. It also creates a new stack frame. The new frame pointer is loaded into AC3.
- Access the arguments to the procedure by indexing off the new frame pointer in AC3. If the procedure was invoked by a CALL statement, the first argument address is -5 off the frame pointer, the second argument address is -6 off the frame pointer, etc. If the procedure was invoked by a function reference, the return value address is -5 off the frame pointer; and the first argument address is -6 off the frame pointer, etc.
- End with a RTN instruction. This restores the old values of AC0, AC1, and AC2. It also restores the old frame pointer and loads it into AC3. It jumps to the instruction in the calling procedure following the EJSR instruction.

Example:

```
;       THIS IS AN ASSEMBLY LANGUAGE PROCEDURE TO BE CALLED
;       AS A FUNCTION FROM A PL/I PROGRAM. ITS PURPOSE IS TO
;       SHIFT LEFT OR RIGHT THE BITS OF A WORD-ALIGNED
;       16-BIT DATA ITEM - EITHER A FIXED BINARY, POINTER
;       OR BIT(16) ALIGNED VALUE.
;
;       ENTRY POINTS:
;               SHL - SHIFT LEFT
;               SHR - SHIFT RIGHT
;       ARGUMENTS:
;               1.      VALUE TO SHIFT (FIXED, POINTER, OR BIT(16) ALIGNED)
;               2.      NUMBER OF BITS TO SHIFT

                .ENT    SHR, SHL        ;NAMES OF FUNCTION PROCEDURES

                .NREL   1               ;SHARED CODE PARTITION

SHL:            SAVE    0               ;PUSH RETURN BLOCK, CREATE NEW FRAME
                LDA     1,@-7,3         ;LOAD ARG 2 INTO AC1
                JMP     COMON           ;JUMP TO LABEL COMON

SHR:            SAVE    0               ;PUSH RETURN BLOCK, CREATE NEW FRAME
                LDA     1,@-7,3         ;LOAD ARG 2 INTO AC1
                NEG     1,1             ;NEGATE ARG 2 FOR RIGHT SHIFT

COMON:          LDA     2,@-6,3         ;LOAD ARG 1 INTO AC2
                LSH     1,2             ;DO THE SHIFT
                STA     2,@-5,3         ;STORE THE SHIFTED VALUE
                RTN                     ;RETURN TO CALLING PROCEDURE

                .END
```

To use the procedures SHL and SHR, the source file which contains the above program must be assembled using the AOS Macroassembler. See the AOS Macroassembler Reference Manual (093-000192). For details on the ECLIPSE instructions, see the Programmer's Reference Manual for your ECLIPSE Line computer.

The external entry constants SHL and SHR must be declared as external entry constants in the PL/I program. And of course, the assembled object file containing SHL and SHR must be bound in with the PL/I program.

Following is an example of the use of SHL and SHR:

```
DECLARE (SHL,SHR)
        ENTRY (BIT16) ALIGNED, FIXED BINARY)
        RETURNS (BIT(16) ALIGNED);

DECLARE WORD BIT(16) ALIGNED;

WORD="FFFF"B4;
PUT LIST (WORD, SHL(WORD, 1), SHR(WORD, 5));
```

OUTPUT: "1111111111111111"B "1111111111111110"B "0000011111111111"B

The arguments are easy to address in the above example (LDA @ through their address) because they are word aligned data items and the addresses pushed on the stack are word pointers. If an argument is byte-aligned (FIXED DECIMAL, PICTURE, or CHARACTER), a byte pointer is pushed as the argument address. Care must be taken to address the argument correctly. If the argument is bit-aligned, the address pushed is the address of a two-word bit-pointer to the argument. If the argument is CHARACTER VARYING, the address is a word pointer to the length word of the data item.

In the case of the RETURNS attribute, the address pushed is always a word pointer to the location of where to store the return value, regardless of the data type of the return value.

End of Appendix

# Appendix H
# ASCII Table

To find the *octal* value of a character, locate the character, and combine the first two digits at the top of the character's column with the third digit in the far left column

**LEGEND:**

Character code in decimal
EBCDIC equivalent hexadecimal code
Character

| | | 10_ |
|---|---|---|
| 0 | 64 | @ |
| | 7C | |

| OCTAL | 00_ | | | 01_ | | | 02_ | | | 03_ | | | 04_ | | | 05_ | | | 06_ | | | 07_ | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 0 | 00 | NUL | 8 | 16 | BS (BACK-SPACE) | 16 | 10 | DLE ↑P | 24 | 18 | CAN ↑X | 32 | 40 | SPACE | 40 | 4D | ( | 48 | F0 | 0 | 56 | F8 | 8 |
| 1 | 1 | 01 | SOH ↑A | 9 | 05 | HT (TAB) | 17 | 11 | DC1 ↑Q | 25 | 19 | EM ↑Y | 33 | 5A | ! | 41 | 5D | ) | 49 | F1 | 1 | 57 | F9 | 9 |
| 2 | 2 | 02 | STX ↑B | 10 | 15 | NL (NEW LINE) | 18 | 12 | DC2 ↑R | 26 | 3F | SUB ↑Z | 34 | 7F | " (QUOTE) | 42 | 5C | * | 50 | F2 | 2 | 58 | 7A | : |
| 3 | 3 | 03 | ETX ↑C | 11 | 0B | VT (VERT. TAB) | 19 | 13 | DC3 ↑S | 27 | 27 | ESC (ESCAPE) | 35 | 7B | # | 43 | 4E | + | 51 | F3 | 3 | 59 | 5E | ; |
| 4 | 4 | 37 | EOT ↑D | 12 | 0C | FF (FORM FEED) | 20 | 3C | DC4 ↑T | 28 | 1C | FS ↑\ | 36 | 5B | $ | 44 | 6B | , (COMMA) | 52 | F4 | 4 | 60 | 4C | < |
| 5 | 5 | 2D | ENQ ↑E | 13 | 0D | RT (RETURN) | 21 | 3D | NAK ↑U | 29 | 1D | GS ↑] | 37 | 6C | % | 45 | 60 | - | 53 | F5 | 5 | 61 | 7E | = |
| 6 | 6 | 2E | ACK ↑F | 14 | 0E | SO ↑N | 22 | 32 | SYN ↑V | 30 | 1E | RS ↑↑ | 38 | 50 | & | 46 | 4B | . (PERIOD) | 54 | F6 | 6 | 62 | 6E | > |
| 7 | 7 | 2F | BEL ↑G | 15 | 0F | SI ↑O | 23 | 26 | ETB ↑W | 31 | 1F | US ↑← | 39 | 7D | ' (APOS) | 47 | 61 | / | 55 | F7 | 7 | 63 | 6F | ? |

| OCTAL | 10_ | | | 11_ | | | 12_ | | | 13_ | | | 14_ | | | 15_ | | | 16_ | | | 17_ | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 64 | 7C | @ | 72 | C8 | H | 80 | D7 | P | 88 | E7 | X | 96 | 79 | ` (GRAVE) | 104 | 88 | h | 112 | 97 | p | 120 | A7 | x |
| 1 | 65 | C1 | A | 73 | C9 | I | 81 | D8 | Q | 89 | E8 | Y | 97 | 81 | a | 105 | 89 | i | 113 | 98 | q | 121 | A8 | y |
| 2 | 66 | C2 | B | 74 | D1 | J | 82 | D9 | R | 90 | E9 | Z | 98 | 82 | b | 106 | 91 | j | 114 | 99 | r | 122 | A9 | z |
| 3 | 67 | C3 | C | 75 | D2 | K | 83 | E2 | S | 91 | 8D | [ | 99 | 83 | c | 107 | 92 | k | 115 | A2 | s | 123 | C0 | { |
| 4 | 68 | C4 | D | 76 | D3 | L | 84 | E3 | T | 92 | E0 | \ | 100 | 84 | d | 108 | 93 | l | 116 | A3 | t | 124 | 4F | \| |
| 5 | 69 | 65 | E | 77 | D4 | M | 85 | E4 | U | 93 | 9D | ] | 101 | 85 | e | 109 | 94 | m | 117 | A4 | u | 125 | D0 | } |
| 6 | 70 | C6 | F | 78 | D5 | N | 86 | E5 | V | 94 | 5F | ↑ or ^ | 102 | 86 | f | 110 | 95 | n | 118 | A5 | v | 126 | A1 | ~ (TILDE) |
| 7 | 71 | C7 | G | 79 | D6 | O | 87 | E6 | W | 95 | 6D | ← or _ | 103 | 87 | g | 111 | 96 | o | 119 | A6 | w | 127 | 07 | DEL (RUBOUT) |

SD-00217 Character code in octal at top and left of charts.

↑ means CONTROL

End of Appendix

# Index

DataGeneral

**Software Documentation Remarks Form**

## How Do You Like This Manual?

Title ______________________________ No. __________

We wrote the book for you, and naturally we had to make certain assumptions about who you are and how you would use it. Your comments will help us correct our assumptions and improve our manuals. Please take a few minutes to respond.

If you have any comments on the software itself, please contact your Data General representative. If you wish to order manuals, consult the Publications Catalog (012-330).

## Who Are You?

☐ EDP Manager
☐ Senior System Analyst
☐ Analyst/Programmer
☐ Operator
☐ Other ______________

What programming language(s) do you use? __________

## How Do You Use This Manual?

*(List in order: 1 = Primary use)*

____ Introduction to the product
____ Reference
____ Tutorial Text
____ Operating Guide
____ ______________

## Do You Like The Manual?

| Yes | Somewhat | No | |
|---|---|---|---|
| ☐ | ☐ | ☐ | Is the manual easy to read? |
| ☐ | ☐ | ☐ | Is it easy to understand? |
| ☐ | ☐ | ☐ | Is the topic order easy to follow? |
| ☐ | ☐ | ☐ | Is the technical information accurate? |
| ☐ | ☐ | ☐ | Can you easily find what you want? |
| ☐ | ☐ | ☐ | Do the illustrations help you? |
| ☐ | ☐ | ☐ | Does the manual tell you everything you need to know? |

## Comments?

*(Please note page number and paragraph where applicable.)*

## From:

Name ______________ Title ______________ Company ______________

Address ______________ Date ______________

SD-00742

FOLD DOWN FIRST FOLD DOWN

FIRST
CLASS
PERMIT
No. 26
Southboro
Mass. 01772

BUSINESS REPLY MAIL

No Postage Necessary if Mailed in the United States

Postage will be paid by:

**Data General Corporation**

Southboro, Massachusetts 01772

ATTENTION: Software Documentation

FOLD UP SECOND FOLD UP

SD-00742A

STAPLE