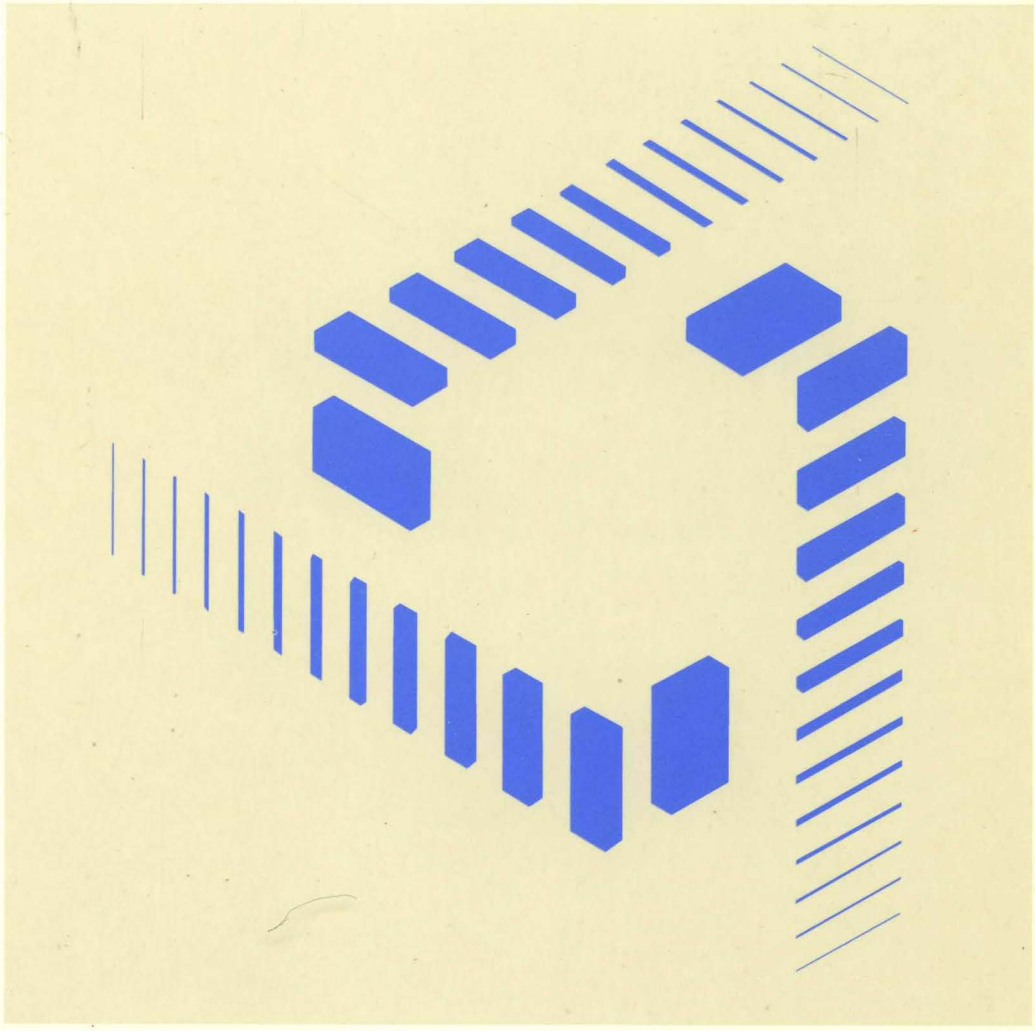
IBM SAA AD/Cycle PL/I Package/2

SC26-4823-00

Language Reference

Release 1



IBM SAA AD/Cycle PL/I Package/2

SC26-4823-00

Language Reference

Release 1

Note!

Before using this information and the product it supports, be sure to read the general information under "Notices" on page xvi.

First Edition (September 1992)

This edition applies to Version 1 Release 1 of IBM SAA AD/Cycle PL/I Package/2, 5601-388, and to any subsequent releases until otherwise indicated in new editions or technical newsletters. Make sure you are using the correct edition for the level of the product.

Order publications through your IBM representative or the IBM branch office serving your locality. Publications are not stocked at the address below.

A form for readers' comments is provided at the back of this publication. If the form has been removed, address your comments to:

IBM Corporation, Department J58
P.O. Box 49023
San Jose, CA, 95161-9023
United States of America

When you send information to IBM, you grant IBM a nonexclusive right to use or distribute the information in any way it believes appropriate without incurring any obligation to you.

Contents

Notices

References in this publication to IBM products, programs, or services do not imply that IBM intends to make these available in all countries in which IBM operates. Any reference to an IBM product, program or service is not intended to state or imply that only that IBM product, program, or service may be used. Any functionally equivalent product, program, or service that does not infringe any of the intellectual property rights of IBM may be used instead of the IBM product, program, or service. The evaluation and verification of operation in conjunction with other products, except those expressly designated by IBM, are the responsibility of the user.

IBM may have patents or pending patent applications covering subject matter in this document. The furnishing of this document does not give you any license to these patents. You can send license inquiries, in writing, to the IBM Director of Commercial Relations, IBM Corporation, Purchase, New York 10577, U.S.A.

Trademarks

The following terms, denoted by an asterisk (*) where they first occur in this publication, are trademarks of the IBM Corporation in the United States or other countries or both:

AD/Cycle
IBM
Language Environment
OS/2
Presentation Manager
PS/2
SAA
Systems Application Architecture

Chapter 1. About this book

Chapter 1. About this book

This book is a reference for programmers using IBM* Systems Application Architecture* AD/Cycle* PL/I Package/2. It is not a tutorial, but is designed for the reader who already has a knowledge of the language and who requires reference information needed to write a program that will be processed by PL/I compiler for OS/2*. It contains guidance information and general-use programming interfaces.

Because this book is a reference manual, it is not intended to be read from front to back, and terms may be used before they are defined. Terms are shown in italics where they are defined in the book, and definitions are indexed.

How PL/I Package/2 fits in the AD/Cycle framework

PL/I Package/2 is a participating product in the AD/Cycle framework, IBM's composite application for developing and maintaining applications in Systems Application Architecture* environments. The AD/Cycle framework consists of tools that support the full range of application development activities, plus an application development platform of specifications and services for integrating those tools. Tools that conform to these specifications and services operate in concert with other conforming tools. They present a common user interface, use common functions when appropriate, and share common application development information.

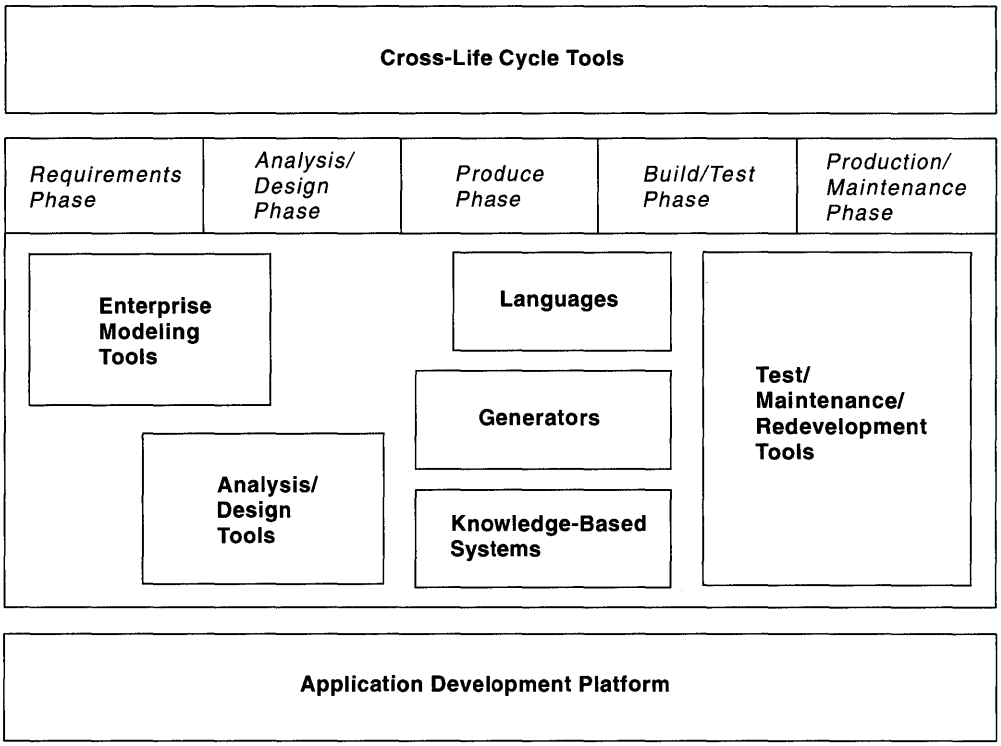


Figure 1. AD/Cycle Framework

AD/Cycle tools are grouped according to the type of application development activities they help perform. Figure 1 shows the AD/Cycle tool sets and their relationship to the traditional development phases. Each tool set is a collection of products, from IBM and members of the IBM International Alliance for the AD/Cycle framework. PL/I Package/2 is part of the languages tool set, indicated by the shaded box. PL/I Package/2 consists of a powerful new language compiler and a Language Environment* that work together to provide solutions for your application development needs.

For more information about the AD/Cycle framework, see the publication *Systems Application Architecture: AD/Cycle Concepts*, GC26-4531.

Using your documentation

The publications provided with PL/I Package/2 are designed to help you do PL/I programming on a personal workstation. Each publication helps you perform a different task.

Where to look for more information

Figure 2. How to Use the Publications You Receive with PL/I Package/2

To...	Use...
Evaluate the product	*PL/I Package/2 Fact Sheet*
Understand warranty information	*PL/I Package/2 Licensed Program Specifications*
Install the compiler and run-time library	*PL/I Package/2 Installation*
Prepare and test your programs and get details on compiler messages	*PL/I Package/2 Programming Guide*
Get details on PL/I syntax and specifications of language elements	*PL/I Package/2 Language Reference* *PL/I Package/2 Reference Summary*
Get details on run-time messages	*PL/I Package/2 Language Environment Run-Time Messages*

For the complete titles and order numbers of these and other related publications, see the "Bibliography" on page 468.

Notation conventions used in this book

The following sections describe how information is presented in this book. Examples and user-supplied information are presented in mixed-case characters.

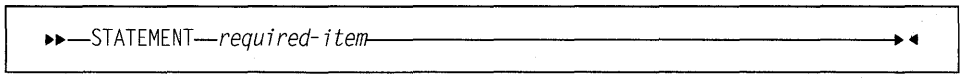
How to read the syntax diagrams

Throughout this book, syntax is described using the following structure:

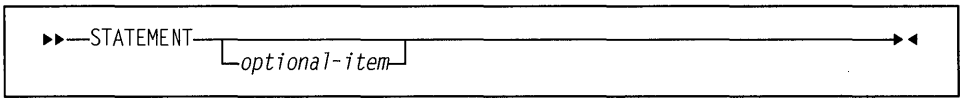
- Read the syntax diagrams from left to right, from top to bottom, following the path of the line. The following table shows the meaning of symbols at the beginning and end of syntax diagram lines.

Symbol	Indicates
▸▸—	the syntax diagram starts here
—▸	the syntax diagram is continued on the next line
▸—	the syntax diagram is continued from the previous line
—▸◂	the syntax diagram ends here

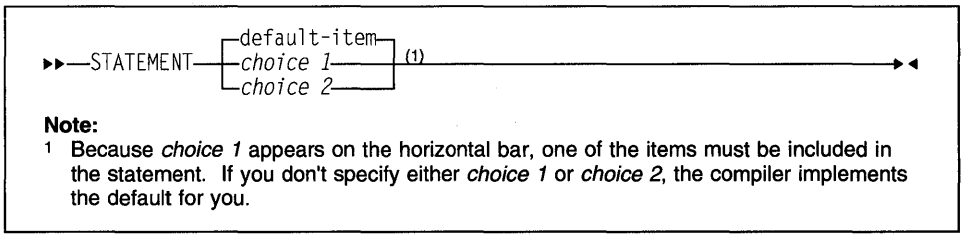
- Required items appear on the horizontal line (the main path).



- Optional items appear below the main path.

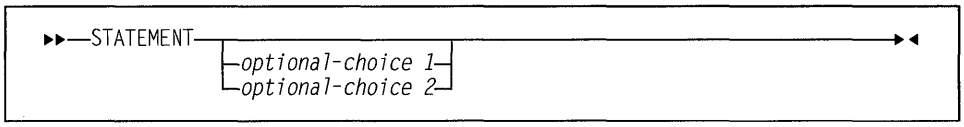


- When you can choose from two or more items, the items appear vertically, in a stack. If you **must** choose one of the items, one item of the stack appears on the main path. The default, if any, appears above the main path and is chosen by the compiler if you do not specify another choice.

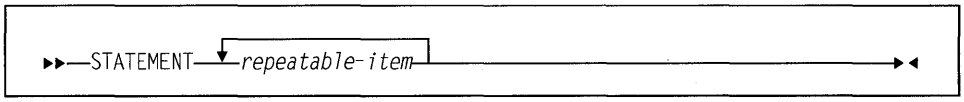


Note:
1 Because *choice 1* appears on the horizontal bar, one of the items must be included in the statement. If you don't specify either *choice 1* or *choice 2*, the compiler implements the default for you.

If choosing one of the items is optional, the entire stack appears below the main path.

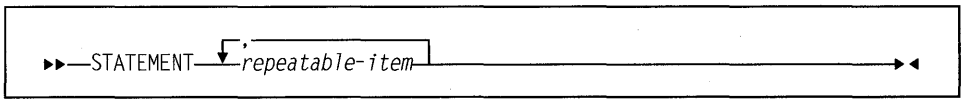


- An arrow returning to the left above the main line is a *repeat arrow*, and it indicates an item that can be repeated.



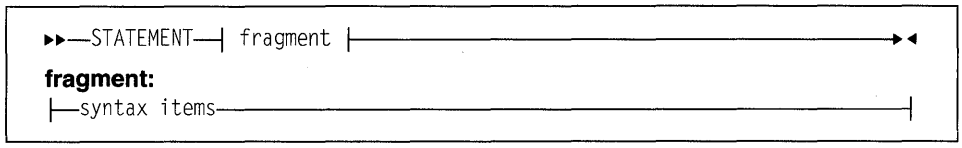
A repeat arrow above a stack indicates that you can make more than one choice from the stacked items, or repeat a single choice.

- If there is a comma as part of the repeat arrow, you must use a comma to separate items in a series.



If the comma appears below the repeat arrow line instead of on the line as shown in the previous example, the comma is optional as a separator of items in a series.

- A syntax fragment is delimited in the main syntax diagram by a set of vertical lines. The corresponding meaning of the fragment begins with the name of the fragment followed by the syntax, which starts and ends with a vertical line.



- Keywords appear in uppercase (for example, STATEMENT) They must be spelled exactly as shown. Variables appear in all lowercase letters (for example, *item*). They represent user-supplied names or values.
- If punctuation marks, parentheses, arithmetic operators, or other symbols are shown, you must enter them as part of the syntax.

Semantics

To describe the PL/I language, the following conventions are used:

- The descriptions are informal. For example, we usually write "x must be a variable" instead of the more precise "x must be the name of a variable." Similarly, we may sometimes write "x is transmitted" instead of "the value of x is transmitted." When the syntax indicates "reference," we may later write "the variable" instead of "the referenced variable."
- When we say that two different source constructs are equivalent, we mean that they produce the same result, and not necessarily that the implementation is the same.
- Unless specifically stated in the text following the syntax specification, the unqualified term "expression" or "reference" refers to a scalar expression. For an expression other than a scalar expression, the type of expression is noted. For example, the term "array expression" indicates that neither a scalar expression nor a structure expression is valid.
- When a result or behavior is *undefined*, it is something you "must not" do. Use of an undefined feature is likely to produce different results on different implementations or releases of a PL/I product. The application program is considered to be in error.
- *Default* is used to describe an alternative value, attribute, or option that is assumed by the system when no explicit choice is specified.
- *Implicit* is used to describe the action taken in the absence of an explicit specification by the program.
- The blank symbol (b) indicates a blank character.

Industry standards used

The PL/I Package/2 compiler compiler is designed according to the specifications of the following industry standards as understood and interpreted by IBM as of December 1987:

- American National Standard Code for Information Interchange (ASCII), X3.4 - 1977
- American National Standard Representation of Pocket Select Characters in Information Interchange, level 1, X3.77 - 1980 (proposed to ISO, March 1, 1979)

- The draft proposed American National Standard Representation of Vertical Carriage Positioning Characters in Information Interchange, level 1, dpANS X3.78 (also proposed to ISO, March 1, 1979)
- Selected features of the American National Standard PL/I General Purpose Subset (ANSI X3.74-1987).

Chapter 2. Program elements

Chapter 2. Program elements

This chapter describes the basic elements that are used to write a PL/I program. The elements include character sets, programmer-defined identifiers, keywords, delimiters, and statements.

PL/I supports the Single Byte Character Set (SBCS) and the Double Byte Character Set (DBCS).

The implementation limits for PL/I's language elements are listed in the Appendix A, "Limits" on page 465.

Single-byte character set

PL/I supports the ASCII Character Set 0850, which contains the English alphabet, digits, special characters, and other national language and control characters. Constants and comments can contain any SBCS character value. PL/I elements (for example, statements, keywords and delimiters) are limited to the characters described in the following sections.

Alphabetic and extralingual characters

The default alphabet for PL/I is the English alphabet plus the extralingual characters.

Alphabetic characters

There are 26 base alphabetic characters that comprise the English alphabet. They are shown in Figure 3 with the equivalent ASCII and EBCDIC values in hexadecimal notation.

Figure 3 (Page 1 of 2). Alphabetic equivalents

Character	EBCDIC Uppercase Hex Value	EBCDIC Lowercase Hex Value	ASCII Uppercase Hex Value	ASCII Lowercase Hex Value
A	C1	81	41	61
B	C2	82	42	62
C	C3	83	43	63
D	C4	84	44	64
E	C5	85	45	65
F	C6	86	46	66
G	C7	87	47	67
H	C8	88	48	68
I	C9	89	49	69
J	D1	91	4A	6A
K	D2	92	4B	6B
L	D3	93	4C	6C
M	D4	94	4D	6D

Figure 3 (Page 2 of 2). Alphabetic equivalents

Character	EBCDIC Uppercase Hex Value	EBCDIC Lowercase Hex Value	ASCII Uppercase Hex Value	ASCII Lowercase Hex Value
N	D5	95	4E	6E
O	D6	96	4F	6F
P	D7	97	50	70
Q	D8	98	51	71
R	D9	99	52	72
S	E2	A2	53	73
T	E3	A3	54	74
U	E4	A4	55	75
V	E5	A5	56	76
W	E6	A6	57	77
X	E7	A7	58	78
Y	E8	A8	59	79
Z	E9	A9	5A	7A

Extralingual characters

The default extralingual characters are the *number* sign (#) the *at* sign (@), and the *currency* sign ($). The hexadecimal values for these characters vary across code pages. You can use the NAMES compiler option to define your own extralingual characters. For more information on defining extralingual characters, refer to the *PL/I Package/2 Programming Guide*.

Alphanumeric characters

An *alphanumeric* character is either an alphabetic or extralingual character, or a digit.

Decimal digits

PL/I recognizes the 10 decimal digits, 0 through 9. They are also known simply as digits and are used to write decimal constants and other representations and values. The following table shows the digits and their hexadecimal values.

Figure 4 (Page 1 of 2). Decimal digit equivalents

Character	EBCDIC Hex Value	ASCII Hex Value
0	F0	30
1	F1	31
2	F2	32
3	F3	33
4	F4	34
5	F5	35
6	F6	36

Figure 4 (Page 2 of 2). Decimal digit equivalents

Character	EBCDIC Hex Value	ASCII Hex Value
7	F7	37
8	F8	38
9	F9	39

Binary digits

PL/I recognizes the 2 binary digits, 0 and 1. They are also known as bits and are used to write binary and bit constants.

Hexadecimal digits

PL/I recognizes the 16 hexadecimal digits, 0 through 9 and A through F. A through F represent the decimal values 10 through 15, respectively. They are also known as hex digits or just hex and are used to write constants in hexadecimal notation.

Special characters

Figure 5 shows the special characters, their PL/I meanings, and their ASCII and EBCDIC values in hexadecimal notation.

Figure 5 (Page 1 of 2). Special character equivalents

Character	Meaning	Default EBCDIC Hex Value	Default ASCII Hex Value
b	Blank	40	20
=	Equal sign or assignment symbol	7E	3D
+	Plus sign	4E	2B
–	Minus sign	60	2D
*	Asterisk or multiply symbol	5C	2A
/	Slash or divide symbol	61	2F
(	Left parenthesis	4D	28
)	Right parenthesis	5D	29
,	Comma	6B	2C
.	Point or period	4B	2E
'	Single quotation mark	7D	27
"	Double quotation mark	7F	22
%	Percent	6C	25
;	Semicolon	5E	3B
:	Colon	7A	3A
¬	Not symbol, exclusive-or symbol (Note 1)	5F	5E
&	And symbol	50	26
|	Or symbol (Note 1)	4F	7C

Figure 5 (Page 2 of 2). Special character equivalents

Character	Meaning	Default EBCDIC Hex Value	Default ASCII Hex Value
>	Greater than symbol	6E	3E
<	Less than symbol	4C	3C
_	Break character (underscore)	6D	5F
?	Question mark	6F	3F

Note 1:

The or (|) and the not (¬) symbols have variant code points. You can use the compiler options OR and NOT to define alternate symbols to represent these operators. For more information about defining symbols, refer to the *PL/I Package/2 Programming Guide*.

Composite symbols

You can combine special characters to create composite symbols. The following table describes these symbols and their meanings. Composite symbols cannot contain blanks.

Figure 6. Composite symbol description

Composite Symbol	Meaning
‖	Concatenation
**	Exponentiation
¬<	Not less than
¬>	Not greater than
¬=	Not equal to
<=	Less than or equal to
>=	Greater than or equal to
/*	Start of a comment
*/	End of a comment
->	Locator

Case sensitivity

You can use a combination of lowercase and uppercase characters in a PL/I program.

When used in keywords or identifiers, the lowercase characters convert to the corresponding uppercase characters. This is true even if you entered a lowercase character as a DBCS character.

When used in a comment or in a character, mixed, or a graphic string constant, lowercase characters remain lowercase.

Statement elements for SBCS

This section describes the elements that make up a PL/I program when using SBCS.

A PL/I statement consists of identifiers, delimiters, operators, and constants. Constants are described in Chapter 3, "Data elements" on page 24.

Identifiers

An *identifier* is a series of characters, not contained in a comment or a constant. Except for P, PIC, and PICTURE, identifiers must be preceded and followed by a delimiter. (P, PIC, and PICTURE identifiers can be followed by a character string.) The first character of an identifier must be one of the alphabetic or extralingual characters. Other characters, if any, can be alphabetic, extralingual, digit or the break (_) character.

Identifiers can be PL/I keywords or programmer-defined names. Because PL/I can determine from the context if an identifier is a keyword, you can use any identifier as a programmer-defined name. There are no *reserved* words in PL/I.

PL/I keywords

A *keyword* is an identifier that has a specific meaning in PL/I. Keywords can specify such things as the action to be taken or the attributes of data. For example, READ, DECIMAL, and ENDFILE are keywords. Some keywords can be abbreviated. The keywords and their abbreviations are shown in uppercase letters.

Programmer-defined names

In a PL/I program, *names* are given to variables and program control data. There are also built-in names, condition names, and generic names. Any identifier can be used as a name. A name can have only one meaning in a program block. For example, the same name cannot be used for both a file and a floating-point variable in the same block.

To improve readability, the break character (_) can be used in a name, such as `Gross_Pay`.

Examples of names are:

```
A                   Rate_of_pay

Record              Loop_3
```

Additional requirements for programmer-defined external names are given in "INTERNAL and EXTERNAL attributes" on page 134.

An asterisk (*) may be used as an identifier name in situations where a name is required but you have no need to refer to it.

Delimiters and operators

Delimiters and *operators* are used to separate identifiers and constants. Figure 7 shows delimiters, and Figure 8 shows operators.

Figure 7. Delimiters

Name	Delimiter	Use
Comment	/* */	The /* and */ enclose commentary (this delimiter includes the /* and the */ and any characters between them.)
Comma	,	Separates elements of a list; precedes the BY NAME option
Period	.	Connects elements of a qualified name; decimal or binary point
Semicolon	;	Terminates a statement
Assignment symbol	=	Indicates assignment
Colon	:	Connects prefixes to statements; connects lower-bound to upper-bound in a dimension attribute; used in RANGE specification of DEFAULT statement
Blank	b	Separates elements
Parentheses	( )	Enclose lists, expressions, iteration factors, and repetition factors; enclose information associated with various keywords
Locator	–>	Denotes locator qualification
Percent	%	Indicates % statements
Single quote	'	Encloses constants (indicates the beginning and end of a constant)
Double quote	"	Encloses constants (indicates the beginning and end of a constant)

Note: Omitting certain symbols can cause errors that are difficult to trace. Common errors are unbalanced quotes, unmatched parentheses, unmatched comment delimiters, and missing semicolons.

Figure 8 (Page 1 of 2). Operators

Operator type	Character(s)	Description
Arithmetic	+	Addition or prefix plus
	–	Subtraction or prefix minus
	*	Multiplication
	/	Division
	**	Exponentiation
Comparison	=	Equal to
	¬=	Not equal to
	<	Less than
	¬<	Not less than
	>	Greater than
	¬>	Not greater than
	<=	Less than or equal to
	>=	Greater than or equal to

Figure 8 (Page 2 of 2). Operators

Operator type	Character(s)	Description
Logical	¬	Not, Exclusive-or
	&	And
	|	Or
String	||	Concatenation

The characters used for delimiters can be used in other contexts. For example, the period is a delimiter when used in name qualification (for example, `Weather.Temperature`), but is a decimal point in an arithmetic constant (for example, `3.14`).

Blanks

You can surround each operator or delimiter with blanks (b). One or more blanks must separate identifiers and constants that are not separated by some other delimiter. Any number of blanks can appear wherever one blank is allowed.

Blanks cannot occur within identifiers, composite symbols, or constants (except in character, mixed, and graphic string constants).

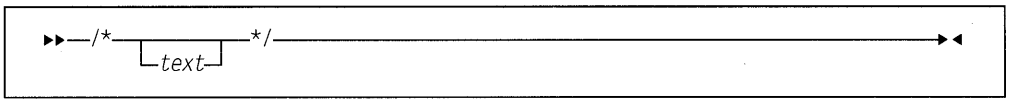
Some examples are:

`ab+bc`	is equivalent to	`Ab + Bc`
`Table(10)`	is equivalent to	`TABLEb(b10bbb)`
`First,Second`	is equivalent to	`first,bsecond`
`AtoB`	is not equivalent to	`AbtobB`

Other cases that require or allow blanks are noted where the feature of the language is discussed.

Comments

Comments are allowed wherever blanks are allowed as delimiters in a program. A comment is treated as a blank and used as a delimiter. Comments are ignored and do not affect the logic of a program. The syntax for a comment is:



/* specifies the beginning of a comment.

text specifies any sequences of characters except the */ composite symbol, which would terminate the comment.

***/** specifies the end of the comment.

A comment can be entered on one or more lines. For example:

```
A = /* This comment is on one line */ 21;

    /* This comment spans
       two lines          */
```

In the following example, what appears to be a comment is actually a character string constant because it is enclosed in quotes.

```
A = '/* This is a constant, not a comment */' ;
```

Statements

You use identifiers, delimiters, operators, and constants to construct PL/I statements.

Although your source program consists of a series of records or lines, PL/I views the program as a continuous stream of characters. There are few restrictions in the format of PL/I statements, and programs can be written without considering special coding rules or checking to see that each statement begins in a specific column. A statement can begin in the next position after the previous statement, or it can be separated by any number of blanks.

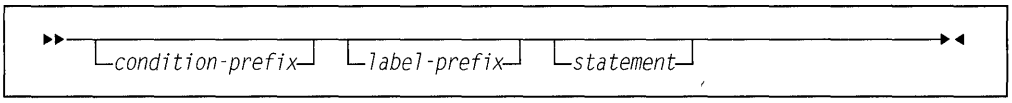
Some statements begin with a % symbol. These statements are either control statements that direct the compilation (controlling listings, including program source text from a library, and so on) or are PL/I Macro Facility statements. A control statement must be on a line by itself.

To improve program readability and maintainability and to avoid unexpected results caused by loss of trailing blanks in source lines, the following recommendations should be followed:

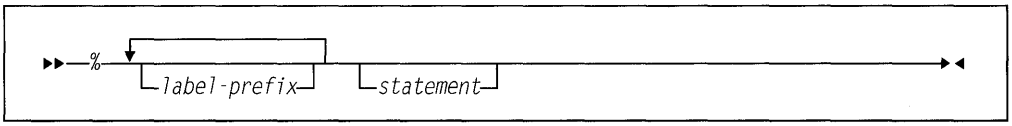
- Do not split a basic language element across lines. If a string constant must be written on multiple lines, use the concatenation operator (||).
- Do not write more than one statement on a line.
- Do not split % control statements across lines.

The PL/I statements and the control statements are alphabetically listed in Chapter 8, "Statements" on page 167. The Macro Facility statements are alphabetically listed in "Macro facility statements" on page 453.

The syntax for a statement is:



The syntax for a % statement is:



Every statement must be contained within some enclosing group or block.

condition-prefix

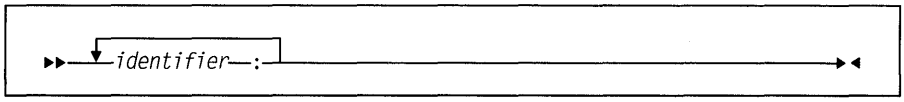
A *condition prefix* specifies the enabling or disabling of a PL/I condition. For more information, refer to Chapter 15, "Condition handling" on page 302.

label-prefix

A *label prefix* is one or more statement labels. It identifies a statement so that it can be referred to at some other point in the program. Statement labels are either label constants (discussed in "Label data and LABEL attribute" on page 46), entry constants (discussed in "Entry data" on page 112), or format constants (discussed in "Format data and FORMAT attribute" on page 47).

Any statement, except DECLARE, DEFAULT, WHEN, OTHERWISE and ON statements, can have a label prefix.

The syntax of a label prefix is:



The syntax for individual statements throughout this book generally does not show the condition prefix or the label prefix.

statement

A simple or a compound statement.

Simple statements

The types of simple statements are: keyword, assignment, and null.

A *keyword statement* is a statement that begins with a keyword. This keyword indicates the function of the statement. In the following example, READ and DECLARE are keywords:

```
READ file(In) into(Input);          /* keyword statement      */
%DECLARE text char;                 /* keyword % statement    */
```

The *assignment statement* contains one or more identifiers on the left side of the assignment symbol (=) and an expression on the right. It does not begin with a keyword:

```
A = B + C;                          /*   assignment statement   */
%size = 15;                         /* % assignment statement   */
```

The *null statement* consists of only a semicolon and is a nonoperational statement:

```
      ;                             /* null statement         */
Label:;                             /* labeled null statement */
    % ;                             /* % null statement       */
```

Compound statements

Compound statements are all keyword statements. Each begins with a keyword which indicates the function of the statement. A compound statement contains one or more simple or compound statements. There are four compound statements: IF, ON, WHEN, and OTHERWISE. A compound statement is terminated by the semicolon that also terminates the final statement of the compound statement.

The following are examples of compound statements:

```
on conversion
  onchar( ) = '0';

if Text = 'stmt' then
  do;
    select(Type);
      when('if') call If_stmt;
      when('do') call do_stmt;
      when('') /* do nothing */ ;
      otherwise
        call Other_stmt;
    end;
    call print;
  end;
end;

%if type = 'AREA' %then
  %size = size + 16;
%else;
```

Groups

Statements can be contained within larger program units called groups. A *group* is either a do-group or a select-group. A do-group is a sequence of statements delimited by a DO statement and a corresponding END statement. A select-group is a sequence of WHEN statements and an OTHERWISE statement delimited by a SELECT statement and a corresponding END statement. The delimiting statements are considered to be part of the group.

When a group is used in a compound statement, control either flows into the group or bypasses it, effectively treating the group as if it were a single statement.

The flow of control within a group is discussed for do-groups under "DO statement" on page 173 and for select-groups under "SELECT statement" on page 193.

Every group must be contained within some enclosing group or block. Groups can contain none, one, or more statements or groups.

Double-byte character set

Each character in the double-byte character set (DBCS) is stored in 2 bytes. DBCS characters can be used anywhere in the source program where a comment, statement label, identifier, or a constant can be used.

When the GRAPHIC compiler option is in effect, source language elements can be written using DBCS and SBCS.

DBCS identifiers

DBCS identifiers can be composed of single-byte characters in DBCS form, double-byte characters, or a combination of both.

Single-byte identifiers in DBCS form

DBCS identifiers containing only single-byte characters must conform to the normal PL/I naming conventions, including the first-character rule. A DBCS identifier containing single-byte characters expressed as DBCS equivalents is a synonym of the same identifier in SBCS.

Notes:

1. This book uses the symbol "." (bold period) to represent the first byte of a double-byte character that can also be represented as SBCS.
2. This book uses "kk" to represent a double-byte character.

Example:

```
.I.B.M = 3;      /* is the same as IBM=3; */
```

DBCS identifiers containing double-byte characters

The sum of the number of SBCS characters plus 2 times the number of DCBS characters in a DBCS name cannot exceed 31. Names containing one or more DBCS characters are considered to be all DBCS. For example,

```
AkkB
Akk.B
.AkkB          /* are all .Akk.B   (3 DBCS characters) */
```

Uses for double-byte character identifiers

A DBCS identifier can be used wherever an SBCS identifier is allowed. When DBCS identifiers are used for EXTERNAL names and %INCLUDE file names, you must ensure that the identifiers are acceptable to the operating system, or are made acceptable by one of the following:

- The EXTERNAL attribute with an environment-name is used.
- The TITLE option of OPEN or FETCH statement is used.

Statement elements for DBCS

This section provides supplemental information about writing PL/I language elements using DBCS. Definitions of the language elements in this section and general usage rules appear in corresponding sections in "Statement elements for SBCS" on page 14. The following is a list of the language elements, an explanation of the rules, and examples of usage.

Identifiers

Use SBCS, DBCS or both.

```
dcl Eof                 /* in SBCS, is the same as        */
dcl .E.o.f              /* this in DBCS.                  */

dcl kkkkX               /* these are all the same, where  */
dcl kkkk.X              /* kk is a valid                  */
dcl kkkkx               /* DBCS character                 */
dcl kkkk.x              /*                                */
```

Comments

Use SBCS, DBCS or both.

```
/* comment */           /* all SBCS                       */
/* .T.y.p.e kk */       /* DBCS text */
```

Mixed Character String Constant

Enclose in SBCS quotes.

Data can be expressed in SBCS or DBCS or both. The DBCS portion is not converted to SBCS.

```
'.a.b.c'M          stored as    .a.b.c          6 bytes
'.I.B.M.'.S'M      stored as    .I.B.M.'.S     10 bytes
'.I.B.M''.S'M      stored as    .I.B.M'.S       9 bytes
'IBMkk'M           stored as    IBMkk           5 bytes
```

Graphic Constant

Enclose in SBCS quotes.

Examples:

```
'.a.b.c'G                /* 6 byte graphic constant  */
'.I.B.M.'.S'G            /* 10 byte graphic constant .I.B.M.'.S  */
```

DBCS continuation rules

If a DBCS character (not a DBCS semicolon) is separated from the right margin by a single SBCS blank, then the blank is ignored and the statement is continued at the left margin of the next input record.

Chapter 3. Data elements

Chapter 3. Data elements

This chapter introduces the kinds of data you can use in PL/I programs and the attributes you use to describe them. The discussion covers:

- A review of data items
- A review of variables and constants
- The types of data that are available and the attributes that define them.

For information on how to declare data, refer to Chapter 7, "Data declaration" on page 128.

Data items

A *data item* is the value of either a variable or a constant. (These terms are not exactly the same as in general mathematical usage. They are discussed further in the next section.) Data items can be single items, called *scalars*, or they can be a collection of items called *data aggregates*.

Data aggregates are groups of data items that can be referred to either collectively or individually. The kinds of data aggregates are *arrays*, *structures*, and *unions*. You can use any type of computational or program control data to form a data aggregate.

Arrays are discussed in "Arrays" on page 144, structures in "Structures" on page 147, unions in "Unions" on page 149, and arrays of structures and unions starting in "Combinations of arrays, structures, and unions" on page 153.

Variables

A *variable* has a value or values that may change during execution of a program. A variable is introduced by a declaration, which declares the name and certain attributes of the variable. A variable having the NONASSIGNABLE attribute is assumed not to change during execution. (Refer to "ASSIGNABLE and NONASSIGNABLE attributes" on page 216 for more information.) A *variable reference* is one of the following:

- A declared variable name
- A reference derived from a declared name through:
  - Pointer qualification
  - Structure qualification
  - Subscripting.

(See Chapter 4, "Expressions and references" on page 50.)

Constants

A *constant* has a value that cannot change. Constants for computational data are referred to by stating the value of the constant or naming the constant in a DECLARE statement. For more information on declaring named constants, see "Named constants" on page 45.

Constants for program control data are referred to by name.

Using quotation marks

String constants, hexadecimal constants, and the picture-specification are enclosed in either single or double quotation marks.

The following rules apply to quotation marks within a string:

- If the included quotation marks are the same type as those used to enclose the string, you must enter two quotation marks (that is, '' or "") for each occurrence to be included.
- If the included quotation marks are the type that is not used to enclose the string, enter only one quotation mark for each instance to be included. The single occurrence is treated as data.

Examples:

```
'Shakespeare''s "Hamlet"' is identical to
"Shakespeare's ""Hamlet"""

PICTURE "99V9" is identical to
PICTURE '99V9'
```

Note: The syntax diagrams in this book show single quotation marks. Double quotation marks can be substituted unless otherwise noted.

Punctuating constants

To improve readability, arithmetic, bit, and hexadecimal constants can use the break character (_). For example:

'1100_1010'B	is the same as	'11001010'B
1100_1010B	is the same as	11001010B
'C_A'B4	is the same as	'ca'b4
'C_A'XN	is the same as	'ca'XN
16_777_216	is the same as	16777216

Data types and attributes

Data used in a PL/I program can be classified as computational data and program control data:

Computational data represents values that are used in computations to produce desired result. It consists of arithmetic and string data.

Arithmetic data is either coded arithmetic data or numeric picture data.

Coded arithmetic data items are rational numbers. They have the data attributes of *base* (BINARY or DECIMAL), *scale* (FLOAT or FIXED), *precision* (significant digits and decimal point placement), and *mode* (REAL or COMPLEX).

Numeric picture data is numeric data that is held in character form and is discussed under "Numeric character data" on page 43.

A *string* is a sequence of contiguous characters, bits, or graphics that are treated as a single data item.

Program control data
represents values that are used to control execution of your program. It consists of the data types: area, entry, label, file, format, pointer, and offset.

For example:

```
Area = (Radius**2) * 3.1416;
```

`Area` and `Radius` are coded arithmetic variables of computational data. The numbers `2` and `3.1416` are coded arithmetic constants of computational data.

If the number `3.1416` is to be used in more than one place in the program, or if it requires specific data or precision attributes, you should declare it as a named constant. Thus, the above statement can be coded as:

```
dcl Pi FIXED DEC (5,4) VALUE(3.1416);
area = (radius**2) * Pi;
```

Constants for program control data have a value that is determined by the compiler. In the following example, the name `loop` represents a label constant of program control data. The value of `loop` is the address of the statement `A=2*B;`.

```
loop:  A=2*B;
       C=B+6;
```

To work with a data item, PL/I needs to know the type of data and how to process it. *Attributes* provide this information. The kinds of attributes are:

Data attributes
describe both computational and program control data items.

AREA	ENTRY	NONVARYING	SIGNED
BINARY	FILE	OFFSET	STRUCTURE
BIT	FIXED	PICTURE	UNSIGNED
CHARACTER	FLOAT	POINTER	UNION
COMPLEX	FORMAT	PRECISION	VARYING
DECIMAL	GRAPHIC	REAL	
DIMENSION	LABEL	RETURNS	

Descriptive attributes
describe both computational and program control data items. They can be specified only in DECLARE statements (including ENTRY and GENERIC descriptor lists), and are the only attributes that can be specified after an asterisk in an ENTRY or GENERIC descriptor list.

ALIGNED	CONNECTED	OPTIONAL
ASSIGNABLE	CONTROLLED	UNALIGNED
BYADDR	NONASSIGNABLE	
BYVALUE	NONCONNECTED	

Non-data attributes
describe non-data elements (for example, built-in functions) or provide additional description for elements that have other data attributes. They can be specified only in DECLARE statements (excluding ENTRY and GENERIC descriptor lists).

ABNORMAL	CONDITION	EXTERNAL	KEYED	PARAMETER
AUTOMATIC	DEFINED	GENERIC	LIKE	POSITION
BASED	DIRECT	INITIAL	NORMAL	STATIC
BUFFERED	ENVIRONMENT	INPUT	OPTIONS	
BUILTIN	EXCLUSIVE	INTERNAL	OUTPUT	

For example, the keyword CHARACTER is a data attribute for the string type of computational data. The keyword FILE is a data attribute for the file type of program control data. The INTERNAL scope attribute specifies that the data item is known only within its declaring block.

The details of how you use keywords and expressions to specify the attributes are in Chapter 7, "Data declaration" on page 128. Briefly, they are:

- Explicitly, using a DECLARE statement
- Contextually, letting PL/I determine them
- By using programmer-defined or language-specified defaults.

Figure 9 on page 28 and Figure 10 on page 29 help you correlate PL/I' variety of attributes with its variety of computational and program control data types. The tables show that the constants and the named constants can only have the indicated data and scope attributes (Figure 9 on page 28). Variables can specify additional attributes (Figure 10 on page 29).

In the example,

```
Area = (Radius**2)*3.1416;
```

the constant 3.1416 is given the attributes:

- DECIMAL because it is not explicitly a binary constant
- FIXED because it is fixed point number
- PRECISION(5,4) - 5 significant digits with 4 to the right of the decimal point
- REAL because it does not have an imaginary part
- INTERNAL and ALIGNED.

(See the "Coded arithmetic" row, and "Data Attributes" and "Scope Attributes" columns of Figure 9 on page 28.)

The constant 1.0 (a decimal fixed-point constant) is different from the constants 1 (another decimal fixed-point constant), '1'B (a bit constant), '1' (a character constant), 1B (binary fixed-point constant), or 1E0 (a decimal floating-point constant).

In the following example, the variable `Pi` has the programmer-defined data attributes of FIXED and DECIMAL with a PRECISION of five digits, four to the right of the decimal point.

```
declare Pi fixed decimal(5,4) initial(3.1416);
```

Because this DECLARE statement contains no other attributes for `Pi`, PL/I applies the defaults for the remaining attributes:

- REAL from the Data Attributes column
- ALIGNED from the Alignment Attributes column
- INTERNAL from the Scope Attributes column
- AUTOMATIC from the Storage Attributes column
- SIGNED from the data attributes column.

(See the coded arithmetic row of Figure 10 on page 29.)

Figure 9 (Page 1 of 2). Classification of attributes by constant types

Constant Type	**Data Attributes** (Notes 1 and 2)	**Scope Attributes** (Notes 1 and 2)
Coded arithmetic	REAL | imaginary FLOAT | FIXED BINARY | DECIMAL PRECISION	internal
Named coded arithmetic	REAL | COMPLEX FLOAT | FIXED BINARY | DECIMAL PRECISION VALUE SIGNED | UNSIGNED	internal
String	BIT | CHARACTER | GRAPHIC (*length*)	internal
Named string	BIT | CHARACTER | GRAPHIC [(*length*)] [VARYING | NONVARYING] VALUE	internal
Named locator	POINTER | OFFSET VALUE	internal
Named picture	PICTURE REAL | COMPLEX VALUE	internal
File(Note 4)	FILE ENVIRONMENT STREAM | RECORD INPUT | OUTPUT | UPDATE SEQUENTIAL | DIRECT BUFFERED | UNBUFFERED(Note 5) KEYED PRINT	INTERNAL | EXTERNAL
Entry(Note 6)	ENTRY [RETURNS]	INTERNAL | EXTERNAL
Format(Note 6)	FORMAT	internal

Figure 9 (Page 2 of 2). Classification of attributes by constant types

Constant Type	Data Attributes (Notes 1 and 2)	Scope Attributes (Notes 1 and 2)
Label(Note 6)	LABEL	internal

Notes:

1. Attributes in this table that appear in uppercase can be explicitly declared. Attributes that are in lower case are implicitly given to the data type.
2. Defaults for data attributes are underlined. Because the data attributes for literal constants are contextual, defaults are not applicable. Named constants and file constants have selectable attributes, so defaults are shown.
3. All constants are internal.
4. File Attributes are described in Chapter 10, "Input and output" on page 228.
5. BUFFERED is the default for SEQUENTIAL files. UNBUFFERED is the default for DIRECT files.
6. Format and label constants, and INTERNAL entry constants cannot be declared in a DECLARE statement.

Figure 10 (Page 1 of 2). Classification of attributes by variable types

Variable Type	Data Attributes	Alignment Attributes	Scope Attributes	Storage Attributes
Area	AREA(*size*)	ALIGNED	INTERNAL | EXTERNAL (INTERNAL is mandatory for AUTOMATIC BASED DEFINED PARAMETER)	AUTOMATIC | STATIC | BASED | CONTROLLED (AUTOMATIC is the default for INTERNAL; STATIC is the default for EXTERNAL) Defined variable: DEFINED [POSITION] Parameter: PARAMETER [CONNECTED | NONCONNECTED] [CONTROLLED] [INITIAL [CALL]] [VARIABLE] [NORMAL | ABNORMAL] ASSIGNABLE | NONASSIGNABLE
Coded arithmetic (Note 1)	REAL | COMPLEX FLOAT | FIXED BINARY | DECIMAL PRECISION [SIGNED | UNSIGNED]	ALIGNED | UNALIGNED		
Entry	ENTRY [RETURNS] [LIMITED]			
File	FILE			
Format	FORMAT			
Label	LABEL			
Locator	POINTER | {OFFSET [(area-variable)]}			
Picture	PICTURE REAL | COMPLEX	ALIGNED | UNALIGNED		
String	BIT | CHARACTER | GRAPHIC [(length)] [VARYING | NONVARYING]			

Arrays: DIMENSION may be added to the declaration of any variable. Refer to "Arrays" on page 144 for more information.

Figure 10 (Page 2 of 2). Classification of attributes by variable types

Variable Type	Data Attributes	Alignment Attributes	Scope Attributes	Storage Attributes
Structures and unions: • For a major structure or union: scope, storage (except INITIAL), alignment, STRUCTURE or UNION, and the LIKE attributes may be specified. • For a member that is a structure or a union: alignment, STRUCTURE or UNION, and the LIKE attributes may be specified. • Members always have the INTERNAL scope attribute. Refer to "Structures" on page 147 and "Unions" on page 149 for more information.				

Notes:

1. Undeclared names, or names declared without a data type, default to coded arithmetic variables. Default attributes are described in "Defaults for attributes" on page 141. Defaults shown are IBM defaults. ANS defaults are FIXED and BINARY rather than FLOAT and DECIMAL.
2. POSITION can be used only with string overlay defining.

Computational data types and attributes

This section describes the data types classified as computational data and the attributes associated with them.

Coded arithmetic data and attributes

Refer to "Data types and attributes" on page 25 for general information about coded arithmetic data.

Syntax for coded arithmetic data is shown in the following diagram:

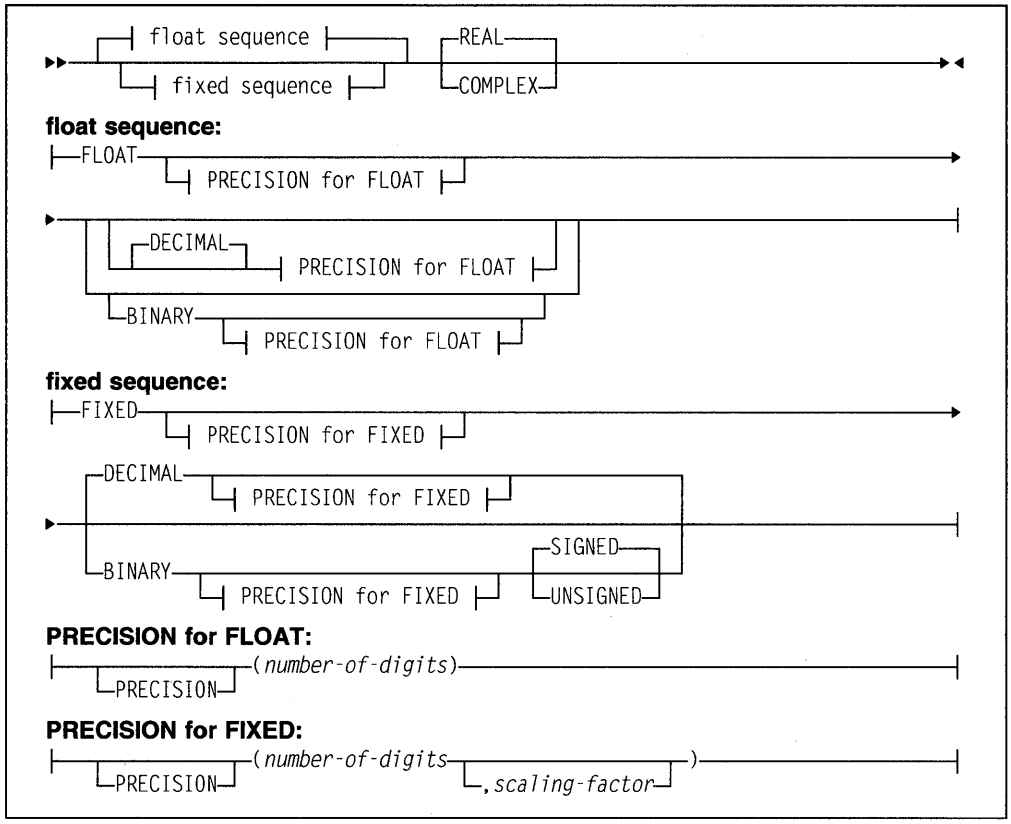


Figure 11. Abbreviations for coded arithmetic data attributes

Attribute	Abbreviation
BINARY	BIN
COMPLEX	CPLX
DECIMAL	DEC
PRECISION	PREC

BINARY and DECIMAL attributes

The *base* of a coded arithmetic data item is either decimal or binary.

Default: DECIMAL.

Refer to "PRECISION attribute" on page 32 for information about the *precision-attribute*.

FIXED and FLOAT attributes

The *scale* of a coded arithmetic data item is either fixed-point or floating-point.

A fixed-point data item is a rational number in which the position of the decimal or binary point is specified, either by its appearance in a constant or by a scaling factor declared for a variable.

Floating-point data items are rational numbers in the form of a fractional part and an exponent part.

PRECISION attribute

The *precision* of a coded arithmetic data item includes the following factors. The scaling factor is used only for floating point items.

number of digits
is an integer that specifies how many digits the value can have. For fixed point items the integer is the number of significant digits. For floating point items the integer is number of significant digits to be maintained excluded the decimal point.

scaling factor
is an optionally-signed integer that specifies the assumed position of the decimal or binary point, relative to the rightmost digit of the number. If no scaling factor is specified, the default is 0.

The precision attribute specification is often represented as (p,q), where *p* represents the number of digits and *q* represents the scaling factor.

A negative scaling factor (-q) specifies an integer, with the point assumed to be located q places to the right of the rightmost actual digit. A positive scaling factor (q) that is larger than the number of digits specifies a fraction, with the point assumed to be located q places to the left of the rightmost actual digit. In either case, intervening zeros are assumed, but they are not stored; only the specified number of digits is actually stored.

If PRECISION is omitted, the precision attribute must follow, with no intervening attribute specifications, the scale (FIXED or FLOAT), base (DECIMAL or BINARY), or mode (REAL or COMPLEX) attributes at the same factoring level.

If included, PRECISION( ) can appear anywhere in the declaration.

Integer value means a fixed-point value with a scaling factor of zero.

REAL and COMPLEX attributes

The *mode* of an arithmetic data item (coded arithmetic or numeric character) is either real or complex.

A real data item is a number that expresses a real value.

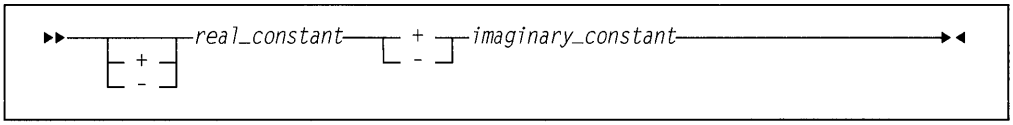
A complex data item consists of two parts—the first a real part and the second an imaginary part. For a variable representing complex data items, the base, scale, and precision of the two parts are identical.

Default REAL for arithmetic variables. Complex arithmetic variables must be explicitly declared with the COMPLEX attribute.

An imaginary constant is written as a real constant of any type immediately followed by the letter I. Examples are:

```
27I
3.968E10I
11011.01BI
```

Each of these has a real part of zero. A complex value with a nonzero real part is represented by an expression with the following syntax.



For example, 38+27I.

Given two complex numbers, y and z:

```
y = complex(A,B);
z = complex(C,D);
```

x=y/z is calculated by:

```
real(x) = (A*C + B*D)/(C**2 + D**2);
imag(x) = (B*C - A*D)/(C**2 + D**2);
```

x=y*z is calculated as follows:

```
real(x) = A*C - B*D;
imag(x) = B*C + A*D;
```

Computational conditions can be raised during these calculations.

SIGNED and UNSIGNED attributes

The SIGNED and UNSIGNED attributes can be used only with FIXED BINARY variables. SIGNED indicates that the variable can assume negative values. UNSIGNED indicates that it can assume only nonnegative values. The UNSIGNED attribute permits the compiler to generate more efficient code in some situations, such as the BY expression in a type 3 DO loop or the second argument to the MOD built-in function.

SIGNED and UNSIGNED have no effect on the semantics of fixed point operations. All intermediate results are SIGNED. For example, the following declaration produces a result of SIGNED FIXED BINARY(8).

```
dcl U7 unsigned fixed bin(7);
call X(U7 + U7);
```

X is, therefore, passed a 2-byte signed integer unless the entry descriptor for X indicates otherwise.

The built-in functions REAL and IMAG, when applied to an unsigned argument, have an unsigned result. All other built-ins, except UNSIGNED, have a signed result.

These attributes also affect the storage requirements, as shown in Figure 12 on page 34 and Figure 13 on page 34.

Figure 12. FIXED BIN SIGNED data storage requirements

This precision:	Occupies this amount of storage:
precision <= 7	1
7 < precision <= 15	2
15 < precision <= 31	4

Figure 13. FIXED BIN UNSIGNED data storage requirements

This precision:	Occupies this amount of storage:
precision <= 8	1
8 < precision <= 16	2
16 < precision <= 31	4

Binary fixed-point data

The data attributes for declaring binary fixed-point variables are BINARY and FIXED. For example:

```
declare Factor binary fixed (20,2);
```

Factor is declared as a variable that can represent binary fixed-point data of 20 data bits, two of which are to the right of the binary point.

Refer to "SIGNED and UNSIGNED attributes" on page 33 for information on how much storage signed and unsigned fixed binary data occupy.

The declared number of data bits is in the low-order positions, but the extra high-order bits participate in any operation performed upon the data item. Any arithmetic overflow into such extra high-order bit positions can be detected only if the SIZE condition is enabled.

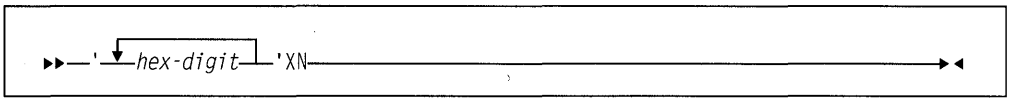
Binary fixed-point constant

A binary fixed-point constant consists of one or more bits with an optional binary point, followed immediately by the letter B. Binary fixed-point constants have a precision (p,q), where *p* is the total number of data bits in the constant, and *q* is the number of bits to the right of the binary point. Examples are:

Constant	Precision
1011_0B	(5,0)
1111_1B	(5,0)
101B	(3,0)
1011.111B	(7,3)

XN (hex) binary fixed-point constant

The XN constant describes a REAL FIXED BINARY(31,0) constant in hexadecimal notation. The specified value may be 1 to 8 hexadecimal digits. The syntax is:



Its hexadecimal value is the given value padded on the left with hex zeros if necessary. For example:

```
'100'XN             /*  same as '00000100'XN with value 256      */
'8000'XN            /*  same as '00008000'XN with value 32,768   */
'FFFF'XN            /*  same as '0000FFFF'XN with value 65,535   */
"ffff_ffff"XN       /*  is the value -1                          */
```

Decimal fixed-point data

The data attributes for declaring decimal fixed-point variables are DECIMAL and FIXED. For example:

```
declare A fixed decimal (5,4);
```

specifies that A represents decimal fixed-point data of 5 digits, 4 of which are to the right of decimal point.

These two examples:

```
declare B fixed (7,0) decimal;
declare B fixed decimal(7);
```

both specify that B represents integers of 7 digits.

```
declare C fixed (7,-2) decimal;
```

specifies that C has a scaling factor of -2. This means that C holds 7 digits that range from -9999999*100 to 9999999*100, in increments of 100.

```
declare D decimal fixed real(3,2);
```

specifies that D represents fixed-point data of 3 digits, 2 of which are fractional.

Decimal fixed-point data is stored two digits per byte, with a sign indication in the rightmost 4 bits of the rightmost byte. Consequently, a decimal fixed-point data item is always stored as an odd number of digits, even though the declaration of the variable may specify the number of digits, *p*, as an even number.

When the declaration specifies an even number of digits, the extra digit place is in the high-order position, and it participates in any operation performed upon the data item, such as in a comparison operation. Any arithmetic overflow or assignment into an extra high-order digit place can be detected only if the SIZE condition is enabled.

Decimal fixed-point constant

A decimal fixed-point constant consists of one or more decimal digits with an optional decimal point. Decimal fixed-point constants have a precision (p,q), where *p* is the total number of digits in the constant and *q* is the number of digits specified to the right of the decimal point. Examples are:

Constant	Precision
3.1416	(5,4)
455.3	(4,1)
732	(3,0)
003	(3,0)
5280	(4,0)
.0012	(4,4)

Binary floating-point data

The data attributes for declaring binary floating-point variables are BINARY and FLOAT. For example:

```
declare S binary float (16);
```

S represents binary floating-point data with a precision of 16 binary digits.

The exponent cannot exceed five decimal digits. Binary floating-point data is stored as IEEE normalized. If the declared precision is less than or equal to (21), short floating-point form is used. If the declared precision is greater than (21) and less than or equal to (53), long floating-point form is used. If the declared precision is greater than (53), extended floating-point form is used.

Binary floating-point constant

A binary floating-point constant is a mantissa followed by an exponent and the letter B. The mantissa is a binary fixed-point constant. The exponent is the letter E, S, D, or Q followed by an optionally-signed decimal integer (meaning 2 to the power of this integer). Constants using E have a precision (p) where *p* is the number of binary digits of the mantissa. Constants using S, D, and Q always have maximum single, double, and extended precisions respectively. Examples are:

Constant	Precision
101101E5B	(6)
101.101E2B	(6)
11101E-28B	(5)
11.01E+42B	(4)
1S0b	(21)
1D0b	(53)
1Q0b	(64)

Decimal floating-point data

The data attributes for declaring decimal floating-point variables are DECIMAL and FLOAT. For example:

```
declare Light_years decimal float(5);
```

in which Light_years represents decimal floating-point data of 5 decimal digits.

Decimal floating-point data is stored as IEEE normalized. If the declared precision is less than or equal to (6), short floating-point form is used. If the declared precision is greater than (6) and less than or equal to (16), long floating-point form is used. If the declared precision is greater than (16), extended floating-point form is used.

Decimal floating-point constant

A decimal floating-point constant is a mantissa followed by an exponent. The mantissa is a decimal fixed-point constant. The exponent is the letter E, S, D, or Q followed by an optionally-signed decimal integer of four or less digits (meaning 10 to the power of this integer). Constants using E have a precision (p) where *p* is the number of digits of the mantissa. Constants using S, D, and Q always represent single, double, and extended precision respectively. Examples are:

Constant	Precision
15E-23	(2)
15E23	(2)
4E-3	(1)
1.96E+07	(3)
438E0	(3)
3_141_593E-6	(7)
.003_141_593E3	(9)
1s0	(6)
1d0	(16)
1q0	(18)

The last two examples represent the same value (although with different precisions).

String data and attributes

Refer to "Data types and attributes" on page 25 for general information about strings.

BIT, CHARACTER, and GRAPHIC attributes

The BIT attribute specifies a bit variable.

The CHARACTER attribute specifies a character variable. Character strings can also be declared using the PICTURE attribute.

The GRAPHIC attribute specifies a graphic variable.

The syntax for the BIT, CHARACTER, and GRAPHIC attributes is:

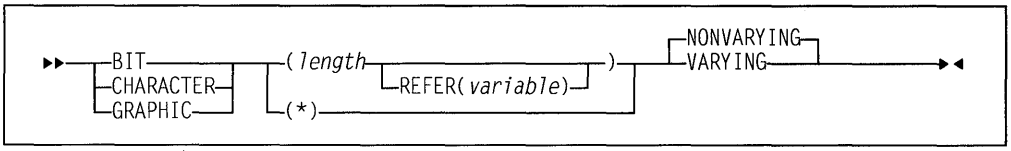


Figure 14. Abbreviations for string data attributes

Attribute	Abbreviation
CHARACTER	CHAR
GRAPHIC	G
NONVARYING	NONVAR
VARYING	VAR

length specifies the length of a NONVARYING string or the maximum length of a VARYING string. The length is in bits, characters, or graphics (DBCS characters), as appropriate.

You can specify the length as an expression or an asterisk. If the length is not specified, the default is 1. For named constants, length is determined from the length of the value expression.

For a parameter, an expression is valid only if it is CONTROLLED. An asterisk specification for a parameter indicates that the length is to be taken from the argument that is passed.

If the length specification is an expression, it is evaluated and converted to FIXED BINARY, which must be positive, when storage is allocated for the variable.

For STATIC data, length must be a restricted expression.

For BASED data, length must be a restricted expression, unless the string is member of a structure or a union and the REFER option is used.

REFER See "REFER Option (Self-Defining Data)" on page 210 for the description of the REFER option.

The statement below declares `User` as a variable that can represent character data with a length of 15:

```
declare User character (15);
```

The following example shows the declaration of a bit variable:

```
declare Symptoms bit (64);
```

VARYING and NONVARYING attributes

The VARYING attribute specifies that a variable can have a length varying from 0 to the declared maximum length. NONVARYING specifies that a variable always has a length equal to the declared length.

The storage allocated for VARYING strings is 2 bytes longer than the declared length. The leftmost 2 bytes hold the string's current length.

The following DECLARE statement specifies that `User` represents varying-length character data with a maximum length of 15:

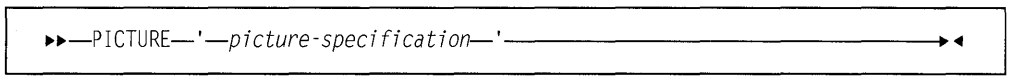
```
declare User character (15) varying;
```

The length for `User` at any time is the length of the data item assigned to it at that time. You can determine the declared and the current length by using the MAXLENGTH and LENGTH built-in functions, respectively.

PICTURE attribute

The PICTURE attribute specifies the properties of a character data item by associating a picture character with each position of the data item. A picture character specifies a category characters that can occupy that position.

The syntax for the PICTURE attribute is:



Abbreviation PIC

picture-specification

describes is either a character data item or a numeric character data item. Refer to "Picture characters for character data" on page 287 or "Picture characters for numeric character data" on page 288 for the valid characters.

An numeric picture specification specifies arithmetic attributes of numeric character data in much the same way that they are specified by the appearance of a constant.

Numeric character data has an arithmetic value but is stored in character form. Numeric character data is converted to coded arithmetic before arithmetic operations are performed.

The base of a numeric character data item is decimal. Its scale is either fixed-point or floating-point (the K or E picture character denotes a floating-point scale). The precision of a numeric character data item is the number of significant digits (excluding the exponent in the case of floating-point). Significant digits are specified by the picture characters for digit positions and conditional digit positions. The scaling factor of a numeric character data item is derived from the V or the F picture character or the combination of V and F.

Only decimal data can be represented by picture characters. Complex data can be declared by specifying the COMPLEX attribute along with a single picture specification that describes either a fixed-point or a floating-point data item.

For more information on numeric character data, see "Numeric character data" on page 43.

Character data

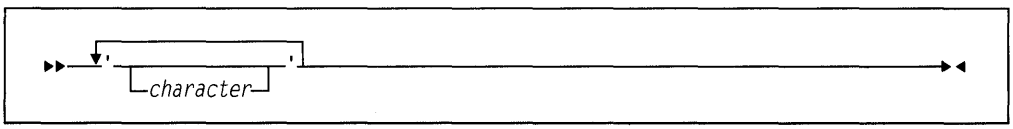
Data with the CHARACTER attribute can contain any of the 256 characters supported by the character set. Data with the PICTURE attribute must have characters that match the picture-specification characters. Each character occupies 1 byte of storage.

Character constant

A character constant is a contiguous sequence of characters enclosed in single or double quotation marks.

Quotation marks included in the constant follow the rules listed in "Using quotation marks" on page 25. The length of a character constant is the number of characters between the enclosing quotation marks counting any doubled quotation marks as a single character.

A null character constant is written as two quotation marks with no intervening blank. The syntax for a character constant is:



Examples of character constants are:

Constant	Length
`'Shakespeare''s "Hamlet"'`	22
`"Shakespeare's ""Hamlet"""`	22
`"Page 5"`	6
`'/* This is not a comment */'`	27
`''`	0
`(2)'Walla '`	12

In the last example, the number in parentheses is a *string repetition factor*, which indicates repetition of the characters that follow. This example is equivalent to the constant `"Walla Walla "`. The string repetition factor must be a constant and enclosed in parentheses.

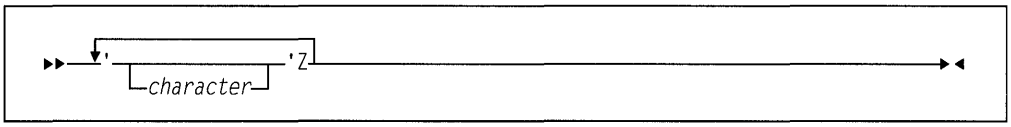
Z (null-terminated) character constant

The Z constant describes a character constant that will be terminated by hexadecimal '00'.

It is written like a character constant but with a Z suffix. The length of a Z constant is the same as a character constant plus 1 for the '00'x supplied by PL/I.

A null Z constant is written as two single or double quotation marks followed by the Z suffix.

The syntax is:



Examples of Z constants are:

Constant	Length
`'Shakespeare''s "Hamlet"'z`	23
`""Z`	1

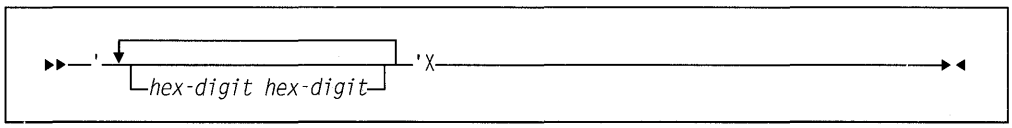
X (hex) character constant

The X character constant is a contiguous sequence of an even number of hex digits enclosed in single or double quotation marks and followed immediately by the letter X. Each pair of hex digits represents one character.

The length of an X constant is half the number of hex digits specified.

A null X constant is written as two quotation marks followed by the X suffix.

The syntax is:



Examples of X constants are:

Constant	Length
"0d0A"x	2
''X	0

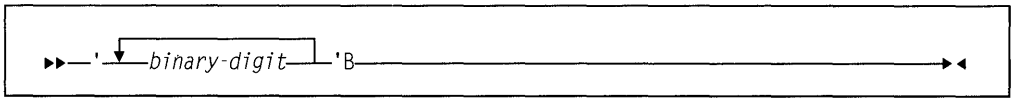
Note: The use of X constant can limit the portability of a program.

Bit data

Data with the BIT attribute permits manipulation of storage in terms of bits. A collection of 8 or fewer unaligned bits occupy 1 byte of storage.

Bit constant

A bit constant is a contiguous sequence binary digits enclosed in single or double quotation marks and followed immediately by the letter B. The syntax for a bit constant is:



A null bit constant is written as two quotation marks, followed by B.

Examples of bit constants are:

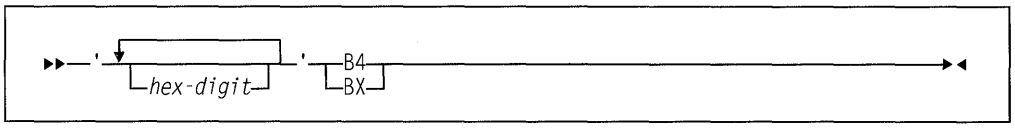
Constant	Length
'1'B	1
"1100_1010_11"B	10
(64)'0'B	64
''B	0
'0'B	1

The number in parentheses in the third example is a string repetition factor which specifies that the following series of bits is to be repeated the specified number of times. The example shown would result in a string of 64 zero bits.

(See "Source-to-target rules" on page 77 for a discussion on the conversion of bit-to-character data and character-to-bit data.)

B4 (hex) bit constant

The B4 bit constant is a contiguous sequence of hex digits enclosed in single or double quotation marks and followed immediately by B4. Each hex digit represents four bits. BX is a synonym for B4. The syntax for a bit constant is:



Some examples of B4 constants are:

```
CA'B4          is the same as      "1100_1010"B
80'B4          is the same as      '1000_0000'B
'1'B4          is the same as      '0001'B
(2)'F'B4       is the same as      '1111_1111'B
(2)'F'B4       is the same as      'FF'BX
'B4            is the same as      ""B
```

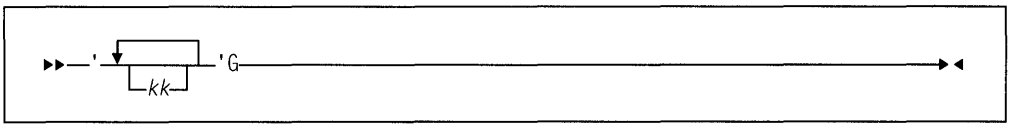
Graphic data

GRAPHIC data can contain any DBCS character. Each DBCS character occupies 2 bytes of storage.

Graphic constant

A graphic constant is a contiguous sequence of DBCS characters enclosed in single or double quotation marks. Graphic constants take up 2 bytes of storage for each DBCS character in the constant.

The syntax for a graphic constant is:

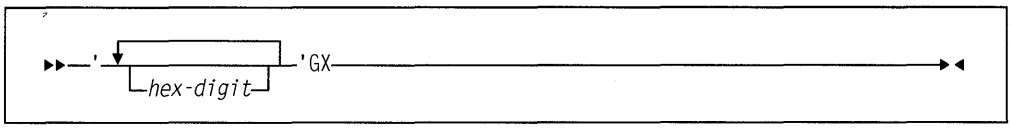


The GRAPHIC compiler option must be in effect for graphic constants to be accepted. If the GRAPHIC ENVIRONMENT option is not specified for STREAM I/O files that include graphic constants, the CONVERSION condition is raised.

GX (hex) graphic constant

The GX graphic constant is a contiguous sequence of hex digits, in multiples of 4, enclosed in single or double quotation marks and followed immediately by GX. Each group of 4 hex digits represents one DBCS character.

The syntax for a GX constant is:



Examples:

```
'81a1'gx        represents one DBCS character
""gX            is the same as ''g
```

Note: The use of GX can limit the portability of a program.

Mixed character data

Mixed character data can contain SBCS and DBCS characters. Mixed data is represented by the CHARACTER data type, and follows the processing rules for CHARACTER data.

The CHARGRAPHIC option of the OPTIONS attribute and the MPSTR built-in function can be used to assist in mixed data handling. For more information on

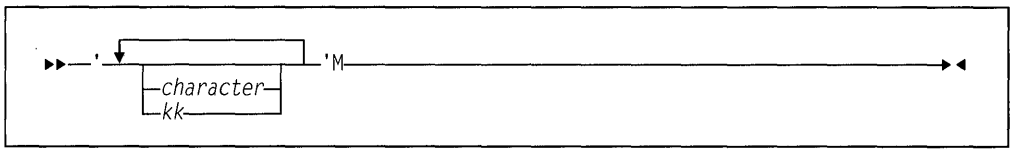
CHARGRAPHIC and MPSTR see "OPTIONS option and attribute" on page 121 and "MPSTR" on page 415.

M (Mixed) character constant

An M constant is a contiguous sequence of DBCS and/or SBCS characters enclosed in quotation marks (single or double), followed immediately by the letter M. Quotations marks included in the constant follow the rules listed in "Using quotation marks" on page 25. The length of an M constant is the number of SBCS characters between the enclosing quotation marks counting any doubled quotation marks as a single character, plus twice the number of DBCS characters in the string.

A null M constant is written as two quotation marks followed by M.

The syntax for the M constant is:



Examples of mixed character constants are:

Constant	Length
'IBM kkkk'M	8 bytes on PS/2*, 10 on S/370
'.I.B.M'M	6 bytes on PS/2, 8 on S/370
''M	0

The GRAPHIC compiler option must be in effect for mixed constants to be accepted. If the GRAPHIC ENVIRONMENT option is not specified for STREAM I/O files having mixed constants, the CONVERSION condition is raised.

Note: Because of the use of shift-codes on some computers, the use of mixed data and M constants can limit program portability.

Numeric character data

A numeric character data item is the value of a variable that has been declared with the PICTURE attribute with a numeric picture specification. The data item is the character representation of a decimal fixed-point or floating-point value.

Numeric picture specification describes a character string that can be assigned only data that can be converted to an arithmetic value.

For example:

```
declare Price picture '999V99';
```

specifies that any value assigned to `Price` is maintained as a character string of five decimal digits, with an assumed decimal point preceding the rightmost two digits. Data assigned to `Price` is aligned on the assumed point in the same way that point alignment is maintained for fixed-point decimal data.

Numeric character data has arithmetic attributes, but it is not stored in coded arithmetic form. Numeric character data is stored as a character string. Before it can be used in arithmetic computations, it must be converted either to decimal fixed-

point or to decimal floating-point format. Such conversions are done automatically, but they require extra processing time.

Although numeric character data is in character form, like character strings, and although it is aligned on the decimal point like coded arithmetic data, it is processed differently from the way either coded arithmetic items or character strings are processed. Editing characters can be specified for insertion into an numeric character data item, and such characters are actually stored within the data item. Consequently, when the item is printed or treated as a character string, the editing characters are included in the assignment. However, if a numeric character item is assigned to another numeric character or arithmetic variable, the editing characters are not included in the assignment—only the actual digits, signs, and the location of the assumed decimal point are assigned. For example:

```
declare Price picture '$99V.99',
        Cost character (6),
        value fixed decimal (6,2);
Price = 12.28;
Cost = '$12.28';
```

In the picture specification for PRICE, the currency symbol ($) and the decimal point (.) are editing characters. They are stored as characters in the data item. However, they are not a part of its arithmetic value. After both assignment statements are executed, the actual internal character representation of `Price` and `Cost` can be considered identical. If they were printed, they would print exactly the same; but they do not always function in the same way. For example:

```
Value = Price;
Cost  = Price;
Value = Cost;
Price = Cost;
```

After the first two assignment statements are executed, the value of `Value` is 0012.28 and the value of `Cost` is `'$12.28'`. In the assignment of `Price` to `Value`, the currency symbol and the decimal point are editing characters, and they are not part of the assignment. The numeric value of `Price` is converted to internal coded arithmetic form. In the assignment of `Price` to `Cost`, however, the assignment is to a character string, and the editing characters of a numeric picture specification always participate in such an assignment. No conversion is necessary because `Price` is stored in character form.

The third and fourth assignment statements would raise the CONVERSION condition. The value of `Cost` cannot be assigned to `Value` because the currency symbol in the string makes it invalid as an arithmetic constant. The value of `Cost` cannot be assigned to `Price` for the same reason. Only values that are of arithmetic type, or that can be converted to arithmetic type, can be assigned to a variable declared with a numeric picture specification.

Although the decimal point can be an editing character or an actual character in a character string, it will not raise the CONVERSION condition in converting to arithmetic form, since its appearance is valid in an arithmetic constant. The same is true for a valid plus or minus sign, because converting to arithmetic form provides for a sign preceding an arithmetic constant.

Other editing characters, including zero suppression characters, drifting characters, and insertion characters, can be used in numeric picture specifications. For a com-

plete discussion of picture characters, see Chapter 14, "Picture specification characters."

Named constants

A named constant is a scalar identifier declared with the VALUE attribute along with other data attributes. All references to the name are logically treated as a reference to the appropriate constant but with the complete set of attributes, whether explicitly declared or defaulted.

Note: The effect of the use of a named constant may not be exactly the same as the use of an unnamed constant. The attributes for a named constant are taken from the declaration which includes explicit and default attributes. The attributes for an unnamed constant are deduced from the shape, form, and size of the constant. For string data, if the length is not specified, or is specified with an asterisk, the length is determined from the length of the restricted expression.

Named constants can be more precise to use in an application program, and they can offer more predictable results. For example, if the named constant `Unit` is defined as FIXED BIN VALUE(1), it has the attributes FIXED BINARY(15) VALUE(1). If you simply use the digit 1, its attributes are FIXED DECIMAL(1,0). See Figure 15 on page 46 for other differences that can occur.

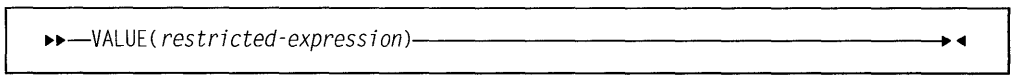
In addition, named constants allow you to set parameters in your application, which makes it easier to debug and maintain.

Named constants can be used wherever a constant is required. They can also be used in restricted expressions that appear later in the program permitting evaluation of a dependent constant.

Named constants can be declared for arithmetic data, string data, and for pointers and offsets. For arithmetic and string data and their attributes, see "String data and attributes" on page 37 and "Coded arithmetic data and attributes" on page 30 respectively. A named constant must be declared before it is used.

VALUE attribute

The syntax for the VALUE attribute is:



restricted expression
the expression must evaluate to a scalar value. For information on restricted expressions see "Restricted expressions" on page 68.

Examples of named constants

Figure 15 shows named constants and the differences in attributes and precisions that can occur between named and unnamed constants.

```
Dcl a4 value(148) fixed bin,
    c4 value(261) fixed bin,
    whole value(800) fixed bin;
Dcl notes (4) static,
    init(a4,  (whole/4),   /* 148, 200  */
         c4,  (whole*2));  /* 261, 1600 */

/* note that "Head" gets length equal to length of VALUE   */

Dcl Head char VALUE('Feel the Power of PL/I'); /* char(22) */
Dcl Headsize fixed bin value(length(Head));     /* 22       */
Dcl 1 Head1 static,
      2 * char(Headsize) initial(Head),        /* char(22) */
      2 * char(20) init(''),
      2 * char(5) init('Page '),
      2 Page_number pic 'zz9',
      2 * char(0);
Dcl TwoHeads char(2*Headsize);                /* char(44) */
Dcl Page0 Picture 'zz9' value(0);
Dcl MyNullPtr ptr value(ptrvalue('ffff_ffff'xn));

/* Differences in attributes/results of
   named and unnamed constants                              */

Dcl pi float bin value (3.1416);  /* is FLOAT BIN(21) but ... */
3.1416                            /* is FIXED DECIMAL(5,4)    */

Dcl Unit fixed bin value(1);      /* is FIXED BIN(15) but ... */
1                                 /* is FIXED DECIMAL(1,0)    */
1.0                               /* is FIXED DECIMAL(2,1)    */
1B                                /* is FIXED BIN(1)          */
0000_0000_0000_001B               /* is FIXED BIN(15)         */

Dcl title char(20) value('SCIDS'); /* is CHAR(20)      but ... */
Dcl title2 char     value('SCIDS');/* is CHAR(5)               */
'SCIDS'                            /* is CHAR(5)               */
```

Figure 15. Named constants

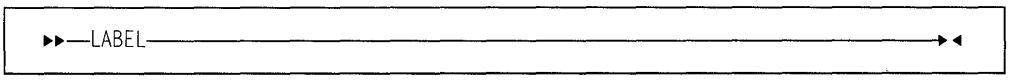
Program control data types and attributes

This section describes program control data and associated attributes. Use program control data to indicate values that control the execution of your program.

Label data and LABEL attribute

A label data item is a label constant or the value of a label variable.

The LABEL attribute specifies that the declared name is a label variable and can have label constants as values.



A label constant is a name written as the label prefix of a statement (other than PROCEDURE, PACKAGE, or FORMAT) so that during execution, program control can be transferred to that statement through a reference to it. ("Statements" on page 17 discusses the syntax of the label prefix.)

In the example:

```
Abcde: Miles = Speed*Hours;
```

Abcde is a label constant. The labelled statement can be executed either by normal sequential execution of instructions or by using the GO TO statement to transfer control to it from some other point in the program.

A label variable can have another label variable or a label constant assigned to it. When such an assignment is made, the environment of the source label is assigned to the target.

A label variable used in a GO TO statement must have as its value a label constant that is used in a block that is active at the time the GO TO is executed. If the variable has an invalid value, the detection of such an error is not guaranteed. For example:

```
declare Lbl_x label;
Lbl_a:   statement;
           ⋮
Lbl_b:   statement;
           ⋮
         Lbl_x = Lbl_a;
             ⋮
         go to Lbl_x;
```

Lbl_a and Lbl_b are label constants, and Lbl_x is a label variable. By assigning Lbl_a to Lbl_x, the statement GO TO Lbl_x transfers control to the Lbl_a statement. Elsewhere, the program can contain a statement assigning Lbl_b to Lbl_x. Then, any reference to Lbl_x would be the same as a reference to Lbl_b. This value of Lbl_x is retained until another value is assigned to it.

In the following example, transfer is made to a particular element of the array Z based on the value of I.

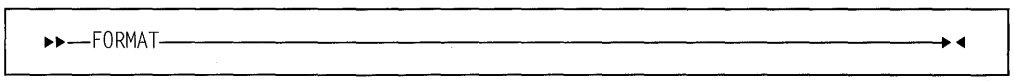
```
go to Z(I);
   ⋮
Z(1):  if X = Y then return;
   ⋮
Z(2):  A = A + B + C * D;
   ⋮
Z(3):  A = A + 10;
```

If Z(2) is omitted, GO TO Z(I) when I=2 raises the ERROR condition. GO TO Z(I) when I < LBOUND(Z) or I > HBOUND(Z) causes unpredictable results if the SUBSCRIPTRANGE condition is disabled.

Format data and FORMAT attribute

A format data item is a format constant or a format variable. A format constant is a name written as the label prefix of a FORMAT statement.

The FORMAT attribute specifies that the name being declared is a format variable. Its syntax is:



A name declared with the FORMAT attribute can have another format variable or a format constant assigned to it. When such an assignment is made, the environment of the source label is assigned to the target.

In the example:

```
Prntexe: format
           ( column(20),A(15), column(40),A(15), column(60),A(15) );
Prntstf: format
           ( column(20),A(10), column(35),A(10), column(50),A(10) );
```

Prntexe and Prntstf are the format constants.

Consider the example:

```
     dcl Print format;
|1|  put edit (X,Y,Z) (R(Prntexe) );
|2|  put edit (X,Y,Z) (R(Prntstf) );
     Print = Prntexe;
|3|  put edit (X,Y,Z) (R(Print) );
     Print = Prntstf;
|4|  put edit (X,Y,Z) (R(Print) );
```

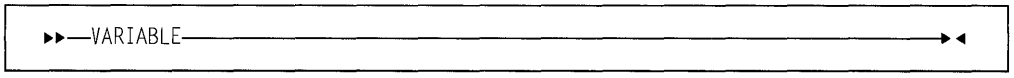
in which |1| and |3| have the same effect, as do |2| and |4|.

VARIABLE attribute

The VARIABLE attribute establishes the name as a variable and is needed only for scalar ENTRY and FILE variables.

Refer to "Entry variables" on page 113 or "File variable" on page 232 for information about these items.

The syntax for the VARIABLE attribute is:



The VARIABLE attribute is implied if the name is a member of a structure or union, or if any of the following attributes is specified:

- Storage class attribute
- DIMENSION
- PARAMETER
- Alignment attribute
- INITIAL

In the following declaration, Account1 and Account2 are file variables and File1 and File2 are file constants.

```
declare Account1 file variable,
        Account2 file automatic,
        File1 file,
        File2 file;
```

File1 and File2 can subsequently be assigned to Account1 or to Account2.

Chapter 4. Expressions and references

Chapter 4. Expressions and references

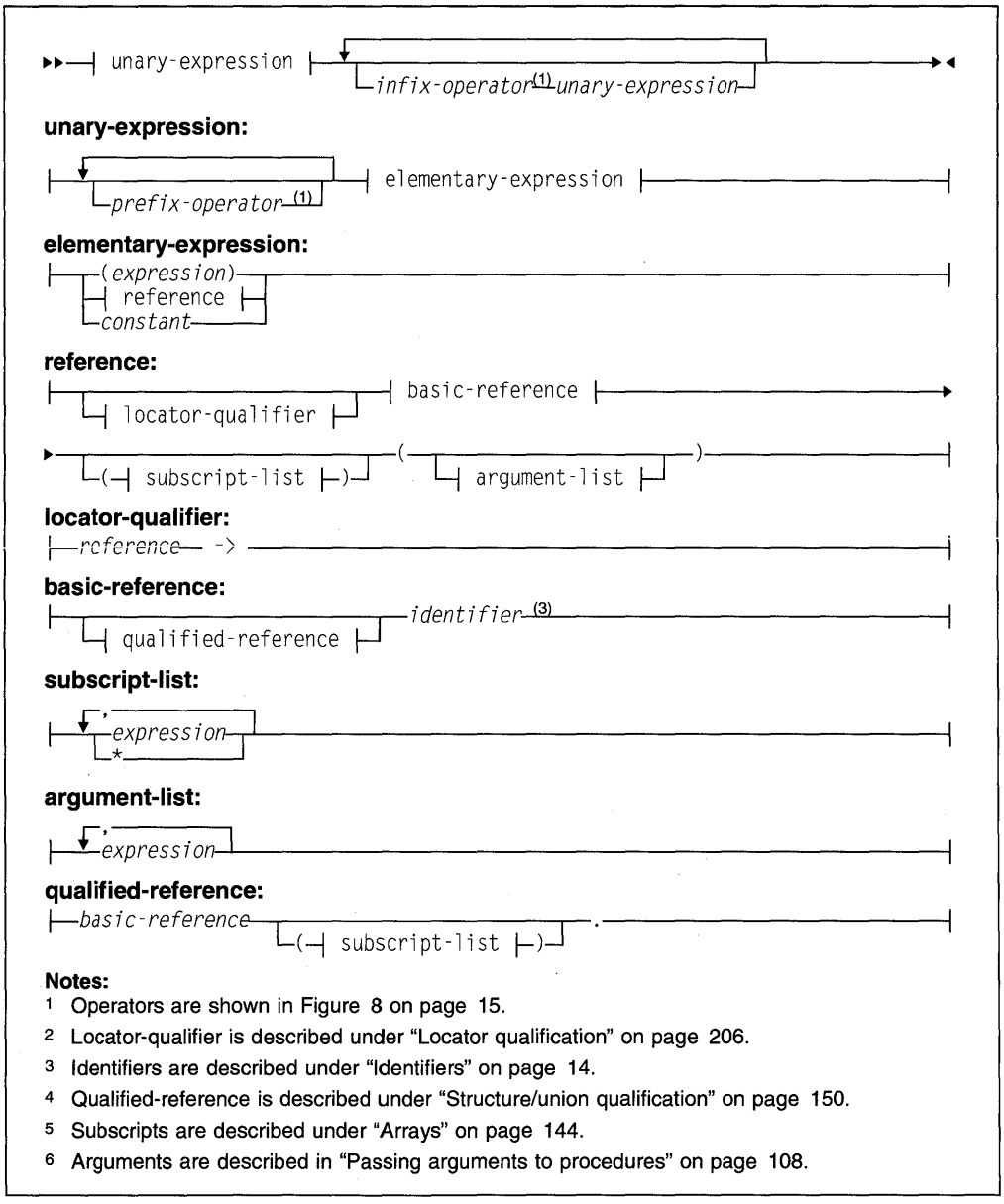
This chapter discusses the various types of expressions and references.

An *expression* is a representation of a value. An expression can be the following:

- A single constant, variable or function reference
- Any combination of constants, variables or function references, including operators and parentheses used in the combination.

An expression that contains operators is an *operational expression*. The constants and variables of an operational expression are called *operands*. See "Operational expressions" on page 53 for more information on operational expressions.

The syntax for expressions and references is:



Notes:

1 Operators are shown in Figure 8 on page 15.
2 Locator-qualifier is described under "Locator qualification" on page 206.
3 Identifiers are described under "Identifiers" on page 14.
4 Qualified-reference is described under "Structure/union qualification" on page 150.
5 Subscripts are described under "Arrays" on page 144.
6 Arguments are described in "Passing arguments to procedures" on page 108.

Any expression can be classified as an *element expression* (also called a scalar expression) or an *array expression*. Element variables and array variables can appear in the same expression.

An element expression
represents an element value. This definition includes an elementary name within a structure or a union or a subscripted name that specifies a single element of an array.

An array expression
represents an array of values. This definition includes a member of a structure or union that has the dimension attribute.

Given the following example:

```
dcl A(10,10) bin fixed(31),
    B(10,10) bin fixed(31),
    1 Rate,
      2 Primary dec fixed(4,2),
      2 Secondary dec fixed(4,2),
    1 Cost(2),
      2 Primary dec fixed(4,2),
      2 Secondary dec fixed(4,2),
    C bin fixed(15),
    D bin fixed(15);
dcl Pi bin float value(3.1416);
```

The following expressions are element expressions:

```
Pi
27
C
C * D
A(3,2) + B(4,8)
Rate.Primary - Cost.Primary(1)
A(4,4) * C
Rate.Secondary / 4
A(4,6) * Cost.Secondary(2)
sum(A)
addr(Rate)
```

The following expressions are array expressions:

```
A
A + B
A * C - D
B / 10B
```

The syntax of many PL/I statements allows expressions, provided the result of the expression conforms with the syntax rules. Unless specifically stated in the text following the syntax specification, the unqualified term *expression* or *reference* refers to a scalar expression. For expressions other than a scalar expression, the type of expression is noted. For example, the term *array expression* indicates that a scalar expression is not valid.

Evaluation order

PL/I statements often contain more than one expression or reference. Except as described for specific instances (for example, the assignment statement), evaluation can be in any order, or (conceptually) at the same time.

For example:

```
dcl (X,Y,Z) entry returns(float), (F,G,H) float;
F = X( Y(G,H), Z(G,H) );
```

The functions Y and Z may change the value of the arguments passed to them. Hence, the value returned by X may be different depending on which function is invoked first. You should not presume that the first parameter is evaluated first. In some situations, it is more optimal to evaluate the last first.

Assuming that the INC function increments the value of the argument passed to it and returns the updated value, the example that follows could put out B(1,2) or B(2,1) depending on which subscript is evaluated first. You should not presume which subscript is evaluated first.

```
dcl B(2,2);
I = 0;
put list ( B( INC(I), INC(I) ) );
```

Targets

The results of an expression evaluation or of a conversion are assigned to a *target*. Targets can be variables, pseudovariables, or intermediate results.

Variables

In the case of an assignment, such as the statement:

```
A = B;
```

the target is the variable on the left of the assignment symbol (in this case A). Assignment to variables can also occur in stream I/O, DO, DISPLAY, and record I/O statements.

Pseudovariables

A pseudovariable represents a target field. For example:

```
declare A character(10),
        B character(30);
substr(A,6,5) = substr(B,20,5);
```

In this assignment statement, the SUBSTR built-in function extracts a substring of length 5 from the string B, beginning with the twentieth character. The SUBSTR pseudovariable indicates the location, within string A, that is the target. Thus, the last 5 characters of A are replaced by characters 20 through 24 of B. The first 5 characters of A remain unchanged.

Pseudovariables can be used only in assignment statements. They are discussed in Chapter 17, "Built-in functions, pseudovariables, and subroutines."

Intermediate results

When an expression is evaluated, the target attributes usually are partly derived from the source, partly from the operation being performed, and partly from the attributes of a second operand. Some defaults may be used, and some implementation restrictions (for example, maximum precision) and conventions exist. An intermediate result may undergo conversion if a further operation is to be performed. After an expression is evaluated, the result may be further converted for assignment to a variable or pseudovariable. These conversions follow the same rules as the conversion of programmer-defined data. For example:

```
declare A character(8),
        B fixed decimal(3,2),
        C fixed binary(10);
A = B + C;
```

During the evaluation of the expression `B + C` and during the assignment of that result, there are four different results:

1. The intermediate result to which the converted binary equivalent of `B` is assigned
2. The intermediate result to which the binary result of the addition is assigned
3. The intermediate result to which the converted decimal fixed-point equivalent of the binary result is assigned
4. `A`, the final destination of the result, to which the converted character equivalent of the decimal fixed-point representation of the value is assigned.

The attributes of the first result are determined from the attributes of the source `B`, from the operator, and from the attributes of the other operand. If one operand of an arithmetic infix operator is binary, the other is converted to binary before evaluation.

The attributes of the second result are determined from the attributes of the source (`C` and the converted representation of `B`).

The attributes of the third result are determined in part from the source (the second result) and in part from the attributes of the eventual target `A`. The only attribute determined from the eventual target is DECIMAL (a binary arithmetic representation must be converted to decimal representation before it can be converted to a character value).

The attributes of `A` are known from the DECLARE statement.

Operational expressions

An operational expression consists of one or more single operations. A single operation is either a *prefix operation* (an operator preceding a single operand) or an *infix operation* (an operator between two operands). The two operands of any infix operation normally should be the same data type when the operation is performed.

The operands of an operation in a PL/I expression are converted, if necessary, to the same data type before the operation is performed. Detailed rules for conversion can be found in Chapter 5, "Data conversion" on page 72.

There are few restrictions on the use of different data types in an expression. However, these mixtures imply conversions. If conversions take place at run time, the program takes longer to run. Also, conversion can result in loss of precision. When using expressions that mix data types, you should understand the relevant conversion rules.

There are five classes of operations—pointer, arithmetic, bit, comparison, and concatenation.

Pointer Operations

The following pointer support extensions can be used.

- Add an expression to or subtract an expression from a pointer expression. The expression type must be computational. If necessary, the nonpointer operand is converted to FIXED BIN(M,0). For example:

  ```
  PTR1 = PTR1 - 16;
  PTR2 = PTR1 + (I*J);
  ```

  You can also use the built-in function, POINTERADD, to perform these operations. You must use POINTERADD if the result is to be used as a locator reference. For example:

  ```
  (PTR1 + 16) -> BASED_PTR          is invalid

  POINTERADD(PTR1,16) -> BASED_PTR  is valid
  ```

- Subtract two pointers to obtain the logical difference. The result is a FIXED BIN(M,0) value. For example:

  ```
  BIN31 = PTR2 - PTR1;
  ```

- Compare pointer expressions using infix operators. For example:

  ```
  IF PTR2 > PTR1 THEN
     BIN31 = PTR2 - PTR1;
  ```

- Use pointer expressions in arithmetic contexts using the built-in function, BINARYVALUE. For example:

  ```
  BIN31 = BIN31 + BINARYVALUE(PTR1);
  ```

- Use computational expressions in pointer contexts using the built-in function, POINTERVALUE. For example:

  ```
  DCL 1 CVTPTR POINTER BASED(POINTERVALUE(16));
  DCL 1 CVT BASED(CVTPTR),
        2 CVT ...;
  ```

  If necessary, the expressions will be converted to FIXED BIN(M,0).

A PL/I block can use pointer arithmetic to access any element within a structure or an array variable. However, the block must be passed the containing structure or array variable, or have the referenced aggregate within its name scope.

Arithmetic operations

An arithmetic operation is specified by combining operands with one of these operators:

```
+   -   *   /   **
```

The plus sign and the minus sign can appear as prefix operators or as infix operators. All other arithmetic operators can appear only as infix operators. (Arithmetic

operations can also be specified by the ADD, SUBTRACT, DIVIDE, and MULTIPLY built-in functions.)

Prefix operators can precede and be associated with any of the operands of an infix operation. For example, in the expression A*-B, the minus sign indicates that the value of A is to be multiplied by -1 times the value of B.

More than one prefix operator can precede and be associated with a single variable. More than one positive prefix operator has no cumulative effect, but two negative prefix operators have the same effect as a single positive prefix operator.

Data conversion in arithmetic operations

The two operands of an arithmetic operation may differ in type, base, mode, precision, and scale. When they differ, conversion takes place as described below. (For coded arithmetic operands, you can also determine conversions using Figure 16 on page 57. Each operand is converted to the type, base, and mode of the result. It is not necessarily converted to the result's precision and scale.)

Note: Scaled FIXED BINARY operands are converted to scaled FIXED DECIMAL before any operations on them are performed.

Type: Character operands are converted to FIXED DECIMAL(N,0). Bit operands are converted to FIXED BIN(M,0). (Refer to Appendix A, "Limits" on page 465 for the maximums.) Numeric character operands are converted to DECIMAL with scale and precision determined by the picture-specification. Graphic variables and strings are allowed in all computational contexts. If conversion is necessary, the rules followed are the same as for character.

The result of an arithmetic operation is always in coded arithmetic form. Type conversion is the only conversion that can take place in an arithmetic prefix operation.

Base: If the bases of the two operands differ, the decimal operand is converted to binary.

Mode: If the modes of the two operands differ, the real operand is converted to complex mode by acquiring an imaginary part of zero with the same base, scale, and precision as the real part. The exception to this is in the case of exponentiation when the second operand (the exponent of the operation) is fixed-point real with a scaling factor of zero. In such a case, conversion is not necessary.

Precision: If only precisions and/or scaling factors vary, type conversion is not necessary.

Scale: If the scales of the two operands differ, the fixed-point operand is converted to floating-point scale. The exception to this is in the case of exponentiation when the first operand is of floating-point scale and the second operand (the exponent of the operation) is fixed-point with a scaling factor of zero, that is, an integer or a variable that has been declared with precision (p,0). In such a case, conversion is not necessary, but the result is floating-point.

If both operands of an exponentiation operation are fixed-point, conversions can occur in one of the following ways:

- Both operands are converted to floating-point if the exponent has a precision other than (p,0).

- The first operand is converted to floating-point unless the exponent is an unsigned integer.
- The first operand is converted to floating-point if precisions indicate that the result of the fixed-point exponentiation would exceed the maximum number of digits allowed.

Results of arithmetic operations

After any necessary conversion of the operands in an expression has been carried out, the arithmetic operation is performed and a result is obtained. This result can be the value of the expression, or it can be an intermediate result upon which further operations are to be performed, or a condition can be raised.

Figure 16 shows the attributes and precisions that result from various arithmetic operations, and Figure 18 shows the attributes of the result for the special cases of exponentiation noted in the right-hand column of Figure 16.

Conversion of operands: In Figure 17 on page 58, if both operands are FIXED and either has a nonzero scale factor, any FIXED BIN operands are converted to FIXED DECIMAL using the following rules:

Given	And	And	The results are
q < 0	p >= q	ceil((p-q/3.32) +9) < N	r = N s = ceil((p-q)/3.32) + q - N
		ceil((p-q)/3.32) + 9) < N	r = ceil((p-q)/3.32) + q s = q
			r = min(N,q) s = r
q < 0		ceil((p-q)/3.32) > N	r = N s = N - ceil((p-q)/3.32)
		ceil((p-q)/3.32) < N	r = ceil((p-q)/3.32) s = 0

For example, FIXED BIN(15,2) is converted to FIXED DEC(6,2). FIXED BIN(15,–2) is converted to FIXED DECIMAL(6,0).

Figure 16. Results of arithmetic operations for one or more FLOAT operands

1st Operand (p1,q1)	2nd Operand (p2,q2)	Attributes of the Result for Addition, Subtraction, Multiplication, or Division	Addition or Subtraction Precision	Multiplication Precision	Division Precision	Attributes of the Result for Exponentiation
FLOAT DECIMAL (p_1)	FLOAT DECIMAL (p_2)	FLOAT DECIMAL (p)	$p = MAX(p_1,p_2)$			FLOAT DECIMAL (p) (unless special case C applies) $p = MAX(p_1,p_2)$
FLOAT DECIMAL (p_1)	FLOAT DECIMAL (p_2,q_2)					
FIXED DECIMAL (p_1,q_1)	FLOAT DECIMAL (p_2)					
FLOAT BINARY (p_1)	FLOAT BINARY (p_2)	FLOAT BINARY (p)				FLOAT BINARY (p) (unless special case C applies) $p = MAX(p_1,p_2)$
FLOAT BINARY (p_1)	FIXED BINARY (p_2,q_2)					
FIXED BINARY (p_1,q_1)	FLOAT BINARY (p_2,q_2)					
FIXED DECIMAL (p_1,q_1)	FLOAT BINARY (p_2)	FLOAT BINARY (p)	$p = MAX(CEIL(p_1*3.32),p_2)$			FLOAT BINARY (p) (unless special case A or C applies) $p = MAX(CEIL(p_1-3.32),p_2)$
FLOAT DECIMAL (p_1)	FLOAT BINARY (p_1,q_2)					
FLOAT DECIMAL (p_1)	FLOAT BINARY (p_2)					
FIXED BINARY (p_1,q_1)	FLOAT DECIMAL (p_2)	FLOAT BINARY (p)	$p = MAX(p_1,CEIL(p_2*3.32))$			FLOAT BINARY (p) (unless special case B or C applies) $p = MAX(p_1,CEIL(p_2-3.32))$
FLOAT BINARY (p_1)	FIXED DECIMAL (p_2,q_2)					
FLOAT BINARY (p_1)	FLOAT DECIMAL (p_2)					

Notes:

1. Special cases of exponentiation are described in Figure 18 on page 59.
2. For a table of CEIL(N*3.32) values, see "Conversion of operands" on page 56.

Figure 17. Results of arithmetic operations between two FIXED operands

1st Operand (p1,q1)	2nd Operand (p2,q2)	Attributes of the Result for Addition, Subtraction, Multiplication, or Division	Addition or Subtraction Precision	Multiplication Precision	Division Precision	Attributes of the Result for Exponentiation
FIXED DECIMAL (p_1,q_1)	FIXED DECIMAL (p_2,q_2)	FIXED DECIMAL (p,q)	p = 1 +MAX(p_1–q_1,p_2–q_2) +q q = MAX(q_1,q_2)	p = p_1+p_2+1 q = q_1+q_2	p = N q = N–p1+q_1–q_2	FLOAT DECIMAL (p) (unless special case A applies) p = MAX(p_1,p_2)
FIXED BINARY (p_1,0)	FIXED BINARY (p_2,0)	FIXED BINARY (p,0)	p = 1 +MAX(p_1–q_1,p_2–q_2) +q q = 0	p = p_1+p_2+1 q = 0	p = M q = 0	FLOAT BINARY (p) (unless special case B applies) p = MAX(p_1,p_2)
FIXED DECIMAL (p_1,0)	FIXED BINARY (p_2,0)	FIXED BINARY (p,0)	p = 1+MAX(r,p_2) q = 0	p = 1+r+p_2 q = 0	p = M q = 0	FLOAT BINARY (p) (unless special case A or C applies) p = MAX(CEIL (p_1–3.32),p_2)
FIXED DECIMAL (p_1,q_1)	FIXED BINARY (p_2,0)	FIXED DECIMAL (p,q)	p = 1 +MAX(p_1–q_1,v) +q q = q_1	p = 1+p_1+v q = q_1	p = N q = N–q_1	
FIXED BINARY (p_1,0)	FIXED DECIMAL (p_2,0)	FIXED BINARY (p,0)	p = 1+MAX(p_1,t) q = 0	p = 1+p_1+t q = 0	p = M q = 0	FLOAT BINARY (p) (unless special case A or C applies) p = MAX(CEIL (p_1–3.32),p_2)
FIXED BINARY (p_1,0)	FIXED DECIMAL (p_2,q_2)	FIXED DECIMAL (p,q)	p = 1 +MAX(p_2–q_2,w) +q q = q_2	p = 1+w+p_2 q = q_1	p = N q = N–q_2	

M is the maximum precision for FIXED BIN.
N is the maximum precision for FIXED DEC.
r = 1 + CEIL(p_1–3.32)
s = CEIL(ABS(q_1–3.32)) * SIGN(q_1)
t = 1 + CEIL(p_2–3.32)
u = CEIL(ABS(q_2*3.32)) * SIGN(q_2)
v = CEIL(p_2/3.32)
w = CEIL(p_1/3.32)

Notes:

The scaling factor must be in the range –128 through +127.

1. Special cases of exponentiation are described in Figure 18 on page 59.
2. For a table of CEIL(N*3.32) values, see Figure 23 on page 76.

Figure 18. Special cases for exponentiation

Case	First Operand	Second Operand	Attributes of Result
A	FIXED DECIMAL (p1,q1)	Integer with value n	FIXED DECIMAL (p,q) (provided p <= N) where p = (p1 + 1)*n–1 and q = q1*n
B	FIXED BINARY (p1,q1)	Integer with value n	FIXED BINARY (p,q) (provided p <= M) where p = (p1 + 1)*n–1 and q = p1*n
C	FLOAT (p1)	FIXED (p2,0)	FLOAT (p1) with base of first operand

Special cases of xy in real/complex modes:**

Real mode:	**Complex mode:**
If x=0 and y>0,	**result is 0**. If x=0, and real part of y>0 and imaginary part of y=0, result is 0.
If x=0 and y<=0,	**ERROR condition is raised**. If x=0 and real part of y<=0 or imaginary part of y ¬=0, ERROR condition is raised.
If x<0 and y not FIXED (p,0),	**ERROR condition is raised**. If x¬=0 and real and imaginary parts of y=0, result is 1.

Consider the expression:

```
A * B + C
```

The operation `A * B` is performed first, to give an intermediate result. Then the value of the expression is obtained by performing the operation (intermediate result) `+ C`.

PL/I gives the intermediate result attributes the same way it gives attributes to any variable. The attributes of the result are derived from the attributes of the two operands (or the single operand in the case of a prefix operation) and the operator involved. The way the attributes of the result are derived is further explained under "Targets" on page 52.

The ADD, SUBTRACT, MULTIPLY, and DIVIDE built-in functions allow you to override the implementation precision rules for addition, subtraction, multiplication, and division operations.

FIXED division: FIXED division can result in overflows or truncation. For example, the result of evaluating the expression:

```
25+1/3
```

is undefined and the FIXEDOVERFLOW condition is raised because FIXED division results in a value of maximum implementation defined precision.

For the following expression, however:

```
25+01/3
```

The result is 25.3333333333333 (when the maximum precision is 15) because constants have the precision with which they are written. The results of the two evaluations are reached as shown in Figure 19:

Figure 19. Comparison of FIXED division and constant expressions

Item	Precision	Result
1	(1,0)	1
3	(1,0)	3
1/3	(15,14)	0.33333333333333
25	(2,0)	25
25+1/3	(15,14)	undefined (truncation on left; FIXEDOVERFLOW is raised when the maximum precision is 15)
01	(2,0)	01
3	(1,0)	3
01/3	(15,13)	00.3333333333333
25	(2,0)	25
25+01/3	(15,13)	25.3333333333333

The PRECISION built-in function can also be used. For example:

```
25+prec(1/3,15,13)
```

Note: Named constants are recommended for situations that require exact precisions.

Bit operations

A bit operation is specified by combining operands with one of the following logical operators:

```
¬   &   |
```

The *not/exclusive-or* symbol (¬), can be used as a prefix or infix operator. The *and* (&) symbol and the *or* (|) symbol, can be used as infix operators only. (The operators have the same function as in Boolean algebra.)

Operands of a bit operation are converted, if necessary, to bit strings before the operation is performed. If the operands of an infix operation do not have the same length, the shorter is padded on the right with '0'B.

The result of a bit operation is a bit string equal in length to the length of the operands.

Bit operations are performed on a bit-by-bit basis. Figure 20 illustrates the result for each bit position for each of the operators. Figure 21 on page 61 shows some examples of bit operations.

Figure 20. Bit operations

A	B	¬A	¬B	A&B	A|B	A¬|B
1	1	0	0	1	1	0
1	0	0	1	0	1	1
0	1	1	0	0	1	1
0	0	1	1	0	0	0

Figure 21. Bit operation examples

For these operands and values	This operation	Yields this result
A = '010111'B B = '111111'B C = '110'B D = 5	¬ A	'101000'B
	¬ C	'001'B
	C & B	'110000'B
	A | B	'111111'B
	A ¬| B	''101000''B
	A ¬| C	'100111'B
	C | B	'111111'B
	A | (¬C)	'011111'B
	¬((¬C)|(¬B))	'110111'B
	SUBSTR(A,1,1)|(D=5)	'1'B

BOOL built-in function

In addition to the *not*, *exclusive-or*, *and*, and *or* operations using the operators ¬, &, and |, Boolean operations can be performed using the BOOL built-in function discussed in "BOOL" on page 387.

Comparison operations

A comparison operation is specified by combining operands with one of the following infix operators:

< ¬< <= = ¬> >= > ¬>

The result of a comparison operation is always a bit string of length 1. The value is '1'B if the relationship is true, or '0'B if the relationship is false.

Note: Scaled FIXED BINARY operands are converted to scaled FIXED DECIMAL before any operations on them are performed. Given a variable with the attributes FIXED BIN(p,q) where q is nonzero, the variable will be converted to FIXED DEC(r,s) where r and s are determined as described in "Conversion of operands" on page 56.

Comparisons are defined as follows:

Algebraic is the comparison of signed arithmetic values in coded arithmetic form. If operands differ in base, scale, precision, or mode, they are converted in a manner analogous to arithmetic operation conversions. Numeric character data is converted to coded arithmetic before comparison. Only the

operators = and ¬= are valid for comparison of operands that are complex numbers.

Character is a left-to-right, character-by-character comparison of characters according to the binary value of the bytes.

Bit is a left-to-right, bit-by-bit comparison of binary digits.

Graphic is a left-to-right, symbol-by-symbol comparison of DBCS characters. The comparison is based on the binary values of the DBCS characters.

Pointer and offset data
is a comparison of pointer and offset values containing any relational operators. However, the only conversion that can take place is offset to pointer.

Program control data
is a comparison of the internal coded forms of the operands. Only the comparison operators = and ¬= are allowed; area variables cannot be compared. No type conversion can take place; all type differences between operands for program control data comparisons are in error.

Comparisons are equal for the following operands:

Entry In a comparison operation, it is not an error to specify an entry variable whose value is an entry point of an inactive block.

Format Format labels on the same FORMAT statement compare equal.

File If the operands represent file values, all of whose parts are equal.

Label Labels on the same statement compare equal. In a comparison operation, it is not an error to specify a label variable whose value is a label constant used in a block that is no longer active.

The label on a compound statement does not compare equal with that on any label contained in the body of the compound statement.

If the operands of a computational data comparison have data types that are appropriate to different types of comparison, the operand of the lower precedence is converted to conform to the comparison type of the other. The precedence of comparison types is (1) algebraic (highest), (2) graphic, (3) character, (4) bit. For example, if a bit string is compared with a fixed decimal value, the bit string is converted to fixed binary for algebraic comparison with the decimal value. The decimal value is also converted to fixed binary.

In the comparison of strings of unequal lengths, the shorter string is padded on the right. This padding consists of:

- Blanks in a character comparison
- '0'B in a bit comparison
- A graphic (DBCS) blank in a graphic comparison.

The following example shows a comparison operation in an IF statement:

```
if A = B
    then action-if-true;
    else action-if-false;
```

The evaluation of the expression A = B yields either '1'B, for true, or '0'B, for false.

In the following assignment statement:

```
X = A <= B;
```

the value '1'B is assigned to X if A is less than B; otherwise, the value '0'B is assigned.

In the following assignment statement:

```
X = A = B;
```

the first equal symbol is the assignment symbol; the second equal symbol is the comparison operator. The value '1'B is assigned to X if A is equal to B; otherwise, the value '0'B is assigned.

An example of comparisons in an arithmetic expression is:

```
(X<0)*A + (0<=X & X<=100)*B + (100<X)*C
```

The value of the expression is A, B, or C and is determined by the value of X.

Concatenation operations

A concatenation operation is specified by combining operands with the concatenation infix operator:

||

Concatenation signifies that the operands are to be joined in such a way that the last character, bit, or graphic of the operand to the left immediately precedes the first character, bit, or graphic of the operand to the right, with nothing intervening.

The concatenation operator can cause conversion to a string type because concatenation can be performed only upon strings—either character, bit, or graphic. The results differ according to the setting of the DEFAULT compiler option:

Results under DEFAULT(IBM)

- If one operand is graphic, the result is graphic.
- If either operand is bit or binary, the result is bit.
- Otherwise the result is character.

For example:

```
dcl B bin(4)  initial(4),
    C bit(1)  initial('1'b);
put skip list (B || C);

/*  Produces '01001'  not 'bbb41'  */
```

Results under DEFAULT(ANS)

- If one operand is graphic, the result is graphic.
- If both operands are bit, the result is bit.
- Otherwise the result is character.

For example:

```
dcl B bin(4)  initial(4),
    C bit(1)  initial('1'b);
put skip list (B || C);

/*  Produces 'bbb41', not '01001'  */
```

The result of a concatenation operation is a string whose length is equal to the sum of the lengths of the two operands, and whose type (that is, character, bit, or graphic) is the same as that of the two operands.

If an operand requires conversion for concatenation, the result depends upon the length of the string to which the operand is converted.

For these operands and values	This operation	Yields this result
A = '010111'B B = '101'B C = 'xy,Z' D = 'aa/BB'	A || B	'010111_101'B
	A || A || B	'010111_010111_101'B
	C || D	'xy,Zaa/BB'
	D || C	'aa/BBxy,Z'
	B || D	'101aa/BB'

In the last example, the bit string '101'B is converted to the character string '101' before the concatenation is performed. The result is a character string.

Combinations of operations

Different types of operations can be combined within the same operational expression. Any combination can be used.

For example:

```
declare result bit(3),
  A fixed decimal(1),
  B fixed binary (3),
  C character(2), D bit(4);
Result = A + B < C & D;
```

Each operation within the expression is evaluated according to the rules for that kind of operation, with necessary data conversions taking place before the operation is performed, as follows:

- The decimal value of A is converted to binary base.
- The binary addition is performed, adding A and B.
- The binary result is compared with the converted binary value of C.
- The bit result of the comparison is extended to the length of the bit variable D, and the & operation is performed.

- The result of the & operation, a bit string of length 4, is assigned to `Result` without conversion, but with truncation on the right.

The expression in this example is evaluated operation-by-operation, from left to right. The order of evaluation, however, depends upon the priority of the operators appearing in the expression.

Priority of operators

The priority of the operators in the evaluation of expressions is shown in Figure 22.

Figure 22. Priority of operations and guide to conversions

Priority	Operator	Type of Operation	Remarks
1	**	Arithmetic	Result is in coded arithmetic form
	prefix +, –	Arithmetic	No conversion is required if operand is in coded arithmetic form
			Operand is converted to FIXED DECIMAL if it is a CHARACTER string or numeric character (PICTURE) representation of a fixed-point decimal number
			Operand is converted to FLOAT DECIMAL if it is a numeric character (PICTURE) representation of a floating-point decimal number
			Operand is converted to FIXED BINARY if it is a BIT string
	prefix ¬	Bit string	All non-BIT data converted to BIT
2	*, /	Arithmetic	Result is in coded arithmetic form
3	infix +, –	Arithmetic	Result is in coded arithmetic form
4	||	Concatenation	Refer to "Results under DEFAULT(ANS)" on page 64 and "Results under DEFAULT(IBM)" on page 63
5	<, ¬<, <=, =, ¬=, >=, >, ¬>	Comparison	Result is always either '1'B or '0'B
6	&	Bit string	All non-BIT data converted to BIT
7		Bit string	All non-BIT data converted to BIT
	infix ¬	Bit string	All non-BIT data converted to BIT

Notes:

1. The operators are listed in order of priority, group 1 having the highest priority and group 7 the lowest. All operators in the same priority group have the same priority. For example, the exponentiation operator ** has the same priority as the prefix + and prefix – operators and the *not* operator ¬.
2. For priority group 1, if two or more operators appear in an expression, the order of priority is right to left within the expression; that is, the rightmost exponentiation or prefix operator has the highest priority, the next rightmost the next highest, and so on. For all other priority groups, if two or more operators in the same priority group appear in an expression, their order or priority is their order left to right within the expression.

The order of evaluation of the expression

```
A + B < C & D
```

is the same as if the elements of the expression were parenthesized as

```
(((A + B) < C) & D)
```

The order of evaluation (and, consequently, the result) of an expression can be changed through the use of parentheses. Expressions enclosed in parentheses are evaluated first, to a single value, before they are considered in relation to surrounding operators.

The above expression, for example, might be changed as follows:

```
(A + B) < (C & D)
```

The value of A converts to fixed-point binary, and the addition is performed, yielding a fixed-point binary result (result_1). The value of C converts to a bit string (if valid for such conversion) and the *and* operation is performed. At this point, the expression is reduced to:

```
result_1 < result_2
```

result_2 is converted to binary, and the algebraic comparison is performed, yielding a bit string of length 1 for the entire expression.

The priority of operators is defined only within operands (or sub-operands). Consider the following example:

```
A + (B < C) & (D || E ** F)
```

In this case, PL/I specifies only that the exponentiation will occur before the concatenation. It does not specify the order of the evaluation of (D||E ** F) in relation to the evaluation of the other operand (A + (B < C)).

Any operational expression (except a prefix expression) must eventually be reduced to a single infix operation. The operands and operator of that operation determine the attributes of the result of the entire expression. In the following example, the & operator is the operator of the final infix operation.

```
A + B < C & D
```

The result of the evaluation is a bit string of length 4.

In the next example, because of the use of parentheses, the operator of the final infix operation is the comparison operator:

```
(A + B) < (C & D)
```

The evaluation yields a bit string of length 1.

Array expressions

Array expressions (unlike array references) are allowed only in assignment statements. If the source is an array expression, then the target must be an array of scalars. Arrays of structures can be assigned to like arrays of structures.

Evaluation of an array expression yields an array result. All operations performed on arrays are performed element-by-element, in row-major order. Therefore, all arrays referred to in an array expression must have the same number of dimensions, and each dimension must be of identical bounds.

Array expressions can include operators (both prefix and infix), element variables, and constants. The rules for combining operations and for data conversion of operands are the same as for element operations.

Prefix operators and arrays

The operation of a prefix operator on an array produces an array of identical bounds. Each element of this array is the result of the operation performed on each element of the original array. For example:

```
If A is the array       5   3  -9
                        1   2   7
                        6   3  -4

then -A is the array   -5  -3   9
                       -1  -2  -7
                       -6  -3   4
```

Infix operators and arrays

Infix operations that include an array variable as one operand can have an element or another array as the other operand.

Array-and-element operations

The result of an expression with an element, an array, and an infix operator is an array with bounds identical to the original array. Each element of the resulting array is the result of the operation between each corresponding element of the original array and the single element. For example:

```
If A is the array          5  10   8
                          12  11   3

then A*3 is the array     15  30  24
                          36  33   9

and 9>A is the array of    1   0   1
bit strings of length 1    0   0   1
```

The element of an array-element operation can be an element of the same array. Consider the following assignment statement:

```
A = A * A(1,2);
```

Again, using the above values for A, the newly assigned value of A would be:

```
   50    100   800
 1200   1100 · 300
```

That is, the value of A(1,2) is fetched again.

Array-and-array operations

If the two operands of an infix operator are arrays, the arrays must have the same number of dimensions, and corresponding dimensions must have identical lower bounds and identical upper bounds. The result is an array with bounds identical to those of the original arrays; the operation is performed upon the corresponding elements of the two original arrays. For example:

```
If A is the array          2   4   3
                           6   1   7
                           4   8   2

and if B is the array      1   5   7
                           8   3   4
                           6   3   1
```

```
then A+B is the array      3   9  10
                          14   4  11
                          10  11   3

and A*B is the array       2  20  21
                          48   3  28
                          24  24   2

and A>B is the array of    1   0   0
bit strings of length 1    0   0   1
                           0   1   1
```

Restricted expressions

Where PL/I requires a (possibly signed) constant, a *restricted expression* can be used. A restricted expression is an expression whose value is calculated at compile time and used as a constant. For example, you can use expressions to define constants required for:

- Extents in static, parameter, and based declarations
- Extents in entry descriptions
- Values and iteration factors to be used in static initialization.

A restricted expression is identical to a normal expression but requires that each operand be:

- A constant or a named constant. A named constant must be declared before it is used.
- A built-in function applied to a restricted expression(s), where the built-in function is from the following categories:
  - String-handling
  - Arithmetic (MAX and MIN must have only two arguments. RANDOM is not allowed.)
  - Mathematical
  - Floating-point inquiry
  - Floating-point manipulation
  - Integer manipulation
  - Precision-handling
  - Array-handling functions DIMENSION, LBOUND, and HBOUND
  - Storage-control functions BINARYVALUE, LENGTH, NULL, OFFSETVALUE, POINTERVALUE, SIZE, STORAGE, and SYSNULL.

Examples

```
dcl Max_names fixed bin value (1000),
    Name_size fixed bin value (30),
    Addr_size fixed bin value (20),
    Addr_lines fixed bin value (4);
dcl 1 Name_addr(Max_names),
      2 Name char(Name_size),
      2 * union,
        3 address char(Addr_lines*Addr_size), /* address    */
        3 addr(Addr_lines) char(Addr_size),
      2 * char(0);
dcl One_Name_addr char(size(Name_addr(1)));   /* 1 name/addr*/
dcl Two_Name_addr char(length(One_Name_addr)
                       *2);                /* 2 name/addrs  */
dcl Name_or_addr char(max(Name_size,Addr_size)) based;

dcl Ar(10) pointer;
dcl Ex     entry( dim(lbound(Ar):hbound(Ar)) pointer);
dcl Identical_to_Ar( lbound(Ar):hbound(Ar) ) pointer;
```

If you change the value of any of the named constants in the example, all of the dependent declarations are automatically re-evaluated.

Chapter 5. Data conversion

Chapter 5. Data conversion

This chapter discusses data conversions for computational data. PL/I converts data when a data item with a set of attributes is assigned to another data item with a different set of attributes. In this chapter, *source* refers to the data item to be converted, and *target* refers to the attributes to which the source is converted. Topics discussed for these data conversions include:

Built-in functions
String lengths
Arithmetic precision
Mode
Source-to-target rules.

Examples of data conversion are included at the end of the chapter.

Data conversion for locator data is discussed in "Locator conversion" on page 205.

Conversion of the value of a computational data item can change its internal representation, precision or mode (for arithmetic values), or length (for string values).

The tables that follow summarize the circumstances that can cause conversion to other attributes.

Case	Target Attributes
Assignment	Attributes of variable on left of assignment symbol
Operand in an expression	Determined by rules for evaluation of expressions
Stream input (GET statement)	Attributes of receiving field
Stream output (PUT statement)	As determined by format list if stream is edit-directed, otherwise character-string
Argument to PROCEDURE	Attributes of corresponding parameter
Argument to built-in function or pseudovariable	Depends on the function or pseudovariable
INITIAL attribute	Other attributes of variable being initialized
RETURN statement expression	Attributes specified in PROCEDURE statement
DO statement, BY, TO, or REPEAT option	Attributes of control variable

The following can cause conversion to character values:

Statement	Option
DISPLAY	
Record I/O	KEYFROM KEY
OPEN	TITLE

The following can cause conversion to a BINARY value:

Statement	Option/Attribute/Reference
DECLARE, ALLOCATE, DEFAULT	length, size, dimension, bound, repetition factor
DELAY	milliseconds
FORMAT (and format items in GET and PUT)	iteration factor w, d, s, p
OPEN	LINESIZE, PAGESIZE
I/O	SKIP, LINE, IGNORE
Most statements	subscript

All attributes for source and target data items (except string length) must be specified at compile time. Conversion can raise one of the following conditions: CONVERSION, OVERFLOW, SIZE, or STRINGSIZE. (Refer to Chapter 16, "Conditions.")

Constants can be converted at compile time as well as at run time. In all cases the conversions are as described here.

More than one conversion might be required for a particular operation. The implementation does not necessarily go through more than one. To understand the conversion rules, it is convenient to consider them as being separate. For example:

```
dcl A fixed dec(3,2) init(1.23);
dcl B fixed bin(15,5);
B = A;
```

In this example, the decimal representation of 1.23 is first converted to binary (11,7), as 1.0011101B. Then precision conversion is performed, resulting in a binary (15,5) value of 1.00111B.

Additional examples of conversion are provided at the end of this chapter.

Built-in functions for computational data conversion

Conversions can take place during expression evaluation, I/O GET and PUT operations, and assignment operations, and between arguments and parameters. Conversions can also be initiated with the following built-in functions:

BINARY	FIXED	REAL
BIT	FLOAT	SIGNED
CHAR	GRAPHIC	UNSIGNED
COMPLEX	IMAG	
DECIMAL	PRECISION	

Each is discussed in Chapter 17, "Built-in functions, pseudovariables, and subroutines."

Each function returns a value with the attribute specified by the function name, performing any required conversions.

With the exception of the conversions performed by the COMPLEX, GRAPHIC, and IMAG built-in functions, assignment to a PL/I variable having the required attributes

can achieve the conversions performed by these built-in functions. However, you might find it easier to use a built-in function than to create a variable solely to carry out a conversion.

Converting string lengths

The source string is assigned to the target string from left to right. If the source string is longer than the target, excess characters, bits, or graphics on the right are ignored, and the STRINGSIZE condition is raised. For fixed-length targets, if the target is longer than the source, the target is padded on the right. If STRINGSIZE is disabled, and the length of the source and/or the target is determined at run time, and the target is too short to contain the source, unpredictable results might occur.

Note: If you use SUBSTR with variables as the parameters, and the variables specify a string not contained in the target, unpredictable results can occur.

Character strings are padded with blanks, bit strings with '0'B, and graphic strings with DBCS blank. For example:

```
declare Subject char(10);
Subject = 'Transformations';
```

Transformations has 15 characters. Therefore, when PL/I assigns the string to Subject, it truncates 5 characters from the right end of the string. This is equivalent to executing:

```
Subject = 'Transforma';
```

The first two of the following statements assign equivalent values to Subject and the last two assign equivalent values to Code:

```
Subject = 'Physics';
Subject = 'Physics   ';
declare Code bit(10);
Code = '110011'B;
Code = '1100110000'B;
```

The following statements do *not* assign equivalent values to Subject:

```
Subject = '110011'B;
Subject = '1100110000'B;
```

When the first statement is executed, the bit constant on the right is first converted to a character string and is then extended on the right with blank characters rather than zero characters. This statement is equivalent to:

```
Subject = '110011bbbb';
```

The second of the two statements requires only a conversion from bit to character type and is equivalent to:

```
Subject = '1100110000';
```

A string value is not extended with blank characters or zero bits when it is assigned to a string variable that has the VARYING attribute. Instead, the length of the target string variable is set to the length of the assigned string. However, truncation will occur if the length of the assigned string exceeds the maximum length declared for the varying-length string variable.

Converting arithmetic precision

When an arithmetic value has the same data attributes, except for precision, as the target, precision conversion is required.

For fixed-point data items, decimal or binary point alignment is maintained during precision conversion. Therefore, padding or truncation can occur on the left or right. If nonzero bits or digits on the left are lost, the SIZE condition is raised.

For floating-point data items, truncation on the right, or padding on the right with zeros, can occur.

Converting mode

If a complex value is converted to a real value, the imaginary part is ignored. If a real value is converted to a complex value, the imaginary part is zero.

Converting other data attributes

Source-to-target rules are given, following this section, for converting data items with the following data attributes:

- Coded arithmetic:
  - FIXED BINARY
  - FIXED DECIMAL
  - FLOAT BINARY
  - FLOAT DECIMAL
- Arithmetic character PICTURE
- CHARACTER
- BIT
- GRAPHIC.

Changes in value can occur in converting between decimal representations and binary representation. In converting between binary and decimal, the factor 3.32 is used as follows:

- n decimal digits convert to CEIL (n*3.32) binary digits.
- n binary digits convert to CEIL (n/3.32) decimal digits.

A table of CEIL values is provided in Figure 23 to calculate these conversions.

*Figure 23. CEIL (n*3.32) and CEIL (n/3.32) values*

n	CEIL (n*3.32)	n	CEIL (n/3.32)
1	4	1-3	1
2	7	4-6	2
3	10	7-9	3
4	14	10-13	4
5	17	14-16	5
6	20	17-19	6
7	24	20-23	7
8	27	24-26	8
9	30	27-29	9
10	34	30-33	10
11	37	34-36	11
12	40	37-39	12
13	44	40-43	13
14	47	44-46	14
15	50	47-49	15
16	53(1)	50-53	16
17	57	54-56	17
18	60	57-59	18
19	64	60-63	19
20	67	64-66	20
21	70	67-69	21
22	74	70-73	22
23	77	74-76	23
24	80	77-79	24
25	83	80-83	25
26	87	84-86	26
27	90	87-89	27
28	93	90-92	28
29	97	93-96	29
30	100	97-99	30
31	103	100-102	31
32	107	103-106	32
33	110	107-109	33
		110-112	34
		113-116	35

Note 1: While ceil(16*3.32) = 54, the value 53 is used. If it were not, a float dec(16), when converted to binary, would have to be converted from long floating-point to extended floating-point (because float bin(54) is represented as extended floating-point).

For fixed-point integer values, conversion does not change the value. For fixed-point fractional values, the factor 3.32 provides only enough digits or bits so that the converted value differs from the original value by less than 1 digit or bit in the rightmost place.

For example, the decimal constant .1, with attributes FIXED DECIMAL (1,1), converts to the binary value .0001B, converting 1/10 to 1/16. The decimal constant .10, with attributes FIXED DECIMAL (2,2), converts to the binary value .0001100B, converting 10/100 to 12/128.

Source-to-target rules

Target: Coded Arithmetic

Source:

FIXED BINARY, FIXED DECIMAL,
FLOAT BINARY, and FLOAT DECIMAL

These are all coded arithmetic data. Rules for conversion between them are given under each data type taken as a target.

Arithmetic character PICTURE

Data first converts to decimal with scale and precision determined by the corresponding PICTURE specification. The decimal value then converts to the base, scale, mode, and precision of the target. See the specific target types of coded arithmetic data using FIXED DECIMAL or FLOAT DECIMAL as the source.

CHARACTER

The source string must represent a valid arithmetic constant or complex expression; otherwise, the CONVERSION condition is raised. The constant can be preceded by a sign and can be surrounded by blanks. The constant cannot contain blanks between the sign and the constant, or between the end of the real part and the sign preceding the imaginary part of a complex expression.

The constant has base, scale, mode, and precision attributes. It converts to the attributes of the target when they are independent of the source attributes, as in the case of assignment. See the specific target types of coded arithmetic data using the attributes of the constant as the source.

If an intermediate result is necessary, as in evaluation of an operational expression, the attributes of the intermediate result are the same as if a decimal fixed-point value of precision had appeared in place of the string. (This allows the compiler to generate code to handle all cases, regardless of the attributes of the contained constant.) Consequently, any fractional portion of the constant might be lost. See the specific target types of coded arithmetic data using FIXED DECIMAL as the source.

It's possible that, during the initial conversion of the character data item to an intermediate fixed decimal number, the value might exceed the default size of the intermediate result. If this occurs, the SIZE condition is raised if it is enabled.

If a character string representing a complex number is assigned to a real target, the complex part of the string is not checked for valid arithmetic characters and CONVERSION cannot be raised, since only the real part of the string is assigned to the target.

A null string gives the value zero; a string of blanks is invalid.

BIT

If the conversion occurs during evaluation of an operational expression, the source bit string is converted to an unsigned value that is FIXED BIN(M,0) See the specific target types of coded arithmetic data using FIXED BINARY as the source.

If the source string is longer than the allowable precision, bits on the left are ignored. If nonzero bits are lost, the SIZE condition is raised.

A null string gives the value zero.

GRAPHIC
Graphic variables and strings are converted to CHARACTER, and then follow the rules for character source described on page 77.

Target: FIXED BINARY (p2,q2)

Source:

FIXED DECIMAL (p1,q1)
The precision of the result is p2 = min(N,1+CEIL(p1*3.32)) and q2=CEIL(ABS(q1*3.32))*SIGN(q1).

FLOAT BINARY (p1)
The precision conversion is as described under "Converting arithmetic precision" on page 75 with p1 as declared or indicated and q1 as indicated by the binary point position and modified by the value of the exponent.

FLOAT DECIMAL (p1)
The precision conversion is the same as for FIXED DECIMAL to FIXED BINARY with p1 as declared or indicated and q1 as indicated by the decimal point position and modified by the value of the exponent.

Arithmetic character PICTURE
CHARACTER
BIT
GRAPHIC
See Target: Coded Arithmetic on page 77.

Target: FIXED DECIMAL (p2,q2)

Source:

FIXED BINARY (p1,q1)
The precision of the result is p2=1+CEIL(p1/3.32) and q2=CEIL(ABS(q1/3.32))*SIGN(q1).

FLOAT BINARY (p1)
The precision conversion is the same as for FIXED BINARY to FIXED DECIMAL with p1 as declared or indicated and q1 as indicated by the binary point position and modified by the value of the exponent.

FLOAT DECIMAL (p1)
The precision conversion is as described under "Converting arithmetic precision" on page 75 with p1 as declared or indicated and q1 as indicated by the decimal point position and modified by the value of the exponent.

Arithmetic character PICTURE
CHARACTER
BIT
GRAPHIC
See Target: Coded Arithmetic on page 77.

Target: FLOAT BINARY (p2)

Source:

FIXED BINARY (p1,q1)
The precision of the result is p2=p1. The exponent indicates any fractional part of the value.

FIXED DECIMAL (p1,q1)
The precision of the result is p2=CEIL(p1*3.32). The exponent indicates any fractional part of the value.

FLOAT DECIMAL (p1)
The precision of the result is p2=CEIL(p1*3.32).

Arithmetic character PICTURE
CHARACTER
BIT
GRAPHIC
See Target: Coded Arithmetic on page 77.

Target: FLOAT DECIMAL (p2)

Source:

FIXED BINARY (p1,q1)
The precision of the result is p2=CEIL(p1/3.32). The exponent indicates any fractional part of the value.

FIXED DECIMAL (p1,q1)
The precision of the result is p2=p1. The exponent indicates any fractional part of the value.

FLOAT BINARY (p1)
The precision of the result is p2=CEIL(p1/3.32).

Arithmetic character PICTURE
CHARACTER
BIT
GRAPHIC
See Target: Coded Arithmetic on page 77.

Target: Arithmetic character PICTURE

The arithmetic character PICTURE data item is the character representation of a decimal fixed-point or floating-point value. The following descriptions for source to arithmetic character PICTURE target show those target attributes that allow assignment without loss of leftmost or rightmost digits.

Source:

FIXED BINARY (p1,q1)
The target must imply:

```
fixed decimal (1+x+q-y,q) or
float decimal (x)
```

where x>=CEIL(p1/3.32), y=CEIL(q1/3.32), and q>=y.

FIXED DECIMAL (p1,q1)
The target must imply:

```
fixed decimal (x+q-q1,q) or
float decimal (x)
```

where x>=p1 and q>=q1.

FLOAT BINARY (p1)
The target must imply:

```
fixed decimal (p,q) or
float decimal (p)
```

where p>=CEIL(p1/3.32) and the values of p and q take account of the range of values that can be held by the exponent of the source.

FLOAT DECIMAL (p1)
The target must imply:

```
fixed decimal (p,q) or
float decimal (p)
```

where p>= p1 and the values of p and q take account of the range of values that can be held by the exponent of the source.

Arithmetic character PICTURE
The implied attributes of the source will be either FIXED DECIMAL or FLOAT DECIMAL. See the respective entries for this target.

CHARACTER
See Target: Coded Arithmetic on page 77.

BIT(n)
The target must imply:

```
fixed decimal (1+x+q,q) or
float decimal (x)
```

where x>=ceil(n/3.32) and q>=0.

GRAPHIC
See Target: Coded Arithmetic on page 77.

Target: CHARACTER

Source:

FIXED BINARY, FIXED DECIMAL,
FLOAT BINARY, and FLOAT DECIMAL

The coded arithmetic value is converted to a decimal constant (preceded by a minus sign if it is negative) as described below. The constant is inserted into an intermediate character string whose length is derived from the attributes of the source. The intermediate string is assigned to the target according to the rules for string assignment.

The rules for coded-arithmetic-to-character-string conversion are also used for list-directed and data-directed output, and for evaluating keys (even for REGIONAL files).

FIXED BINARY (p1,q1)

The binary precision (p1,q1) is first converted to the equivalent decimal precision (p,q), where p=1+CEIL(p1/3.32) and q=CEIL(ABS(q1/3.32))*SIGN(q1). Thereafter, the rules are the same as for FIXED DECIMAL to CHARACTER.

FIXED DECIMAL (p1,q1)

If p1>=q1>=0 then:

- The constant is right adjusted in a field of width p1+3. (The 3 is necessary to allow for the possibility of a minus sign, a decimal or binary point, and a leading zero before the point.)
- Leading zeros are replaced by blanks, except for a single zero that immediately precedes the decimal point of a fractional number. A single zero also remains when the value of the source is zero.
- A minus sign precedes the first digit of a negative number. A positive value is unsigned.
- If q1=0, no decimal point appears; if q1>0, a decimal point appears and the constant has q fractional digits.

If p1<q1 or q1<0, a scaling factor appends to the right of the constant; the constant is an optionally-signed integer. The scaling factor appears even if the value of the item is zero. The scaling factor has the syntax:

```
F{+|-}nn
```

where {+|-}nnn has the value of -q1.

The length of the intermediate string is p1+k+3, where k is the number of digits necessary to hold the value of q1 (not including the sign or the letter F).

If the arithmetic value is complex, the intermediate string consists of the imaginary part concatenated to the real part. The left-hand, or real, part is generated as a real source. The right-hand, or imaginary, part is always signed, and it has the letter I appended. The generated string is a complex expression with no blanks between its elements. The length of the intermediate string is:

```
2*p1+7     for p1>=q1>=0
2*(p1+k)+7 for p1>q1 or q1>0
```

The following examples show the intermediate strings that are generated from several real and complex fixed-point decimal values:

```
Precision     Value      String

(5,0)         2947       'bbbb2947'
(4,1)         -121.7     'b-121.7'
(4,-3)        -3279000   '-3279F+3'
(2,1)         1.2+0.3I   'bbb1.2+0.3I'
```

FLOAT BINARY (p1)

The floating-point binary precision (p1) first converts to the equivalent floating-point decimal precision (p), where p=CEIL(p1/3.32). Thereafter, the rules are the same as for FLOAT DECIMAL to CHARACTER.

FLOAT DECIMAL (p1)

A decimal floating-point source converts as if it were transmitted by an E-format item of the form E(w,d,s) where:

w, the length of the intermediate string, is p1+6.

d, the number of fractional digits, is p1-1.

s, the number of significant digits, is p1.

If the arithmetic value is complex, the intermediate string consists of the imaginary part concatenated to the real part. The left-hand, or real, part is generated as a real source. The right-hand, or imaginary, part is always signed, and it has the letter I appended. The generated string is a complex expression with no blanks between its elements. The length of the intermediate string is 2*p+13.

The following examples show the intermediate strings that are generated from several real and complex floating-point decimal values:

```
Precision     Value          String

(5)           1735*10**5     'b1.7350E+08'
(5)           -.001663       '-1.6630E-03'
(3)           1              'b1.00E+00'
(5)           17.3+1.5I      'b1.7300E+01+1.5000E+00I'
```

Arithmetic character PICTURE

A real arithmetic character field is interpreted as a character string and assigned to the target string according to the rules for converting string lengths. If the arithmetic character field is complex, the real and imaginary parts are concatenated before assignment to the target string. Insertion characters are included in the target string.

BIT

Bit 0 becomes the character 0 and bit 1 becomes the character 1. A null bit string becomes a null character string. The generated character string is assigned to the target string according to the rules for converting string lengths.

GRAPHIC

DBCS to SBCS conversion is possible only if there is a corresponding SBCS character. Otherwise, a CONVERSION condition is raised.

Target: BIT

Source:

FIXED BINARY, FIXED DECIMAL,
FLOAT BINARY, and FLOAT DECIMAL

If necessary, the arithmetic value converts to binary and both the sign and any fractional part are ignored. (If the arithmetic value is complex, the imaginary part is also ignored.) The resulting binary value is treated as a bit string. It is assigned to the target according to the rules for string assignment.

FIXED BINARY (p1,q1)

The length of the intermediate bit string is given by:

```
min(n,(p1-q1))
```

If (p1-q1) is negative or zero, the result is a null bit string.

The following examples show the intermediate strings that are generated from several fixed-point binary values:

```
Precision   Value    String

(1)           1      '1'B
(3)          -3      '011'B
(4,2)      1.25      '01'B
```

FIXED DECIMAL (p1,q1)

The length of the intermediate bit string is given by:

```
min(n,CEIL((p1-q1)*3.32))
```

If (p1-q1) is negative or zero, the result is a null bit string.

The following examples show the intermediate strings that are generated from several fixed-point decimal values:

```
Precision   Value   String

(1)           1     '0001'B
(2,1)       1.1     '0001'B
```

FLOAT BINARY (p1)

The length of the intermediate bit string is given by:

```
min(n,p1)
```

FLOAT DECIMAL (p1)

The length of the intermediate bit string is given by:

```
min(n,ceil(p1*3.32))
```

Arithmetic character PICTURE

Data is first interpreted as decimal with scale and precision determined by the corresponding PICTURE specification. The item then converts according to the rules given for FIXED DECIMAL or FLOAT DECIMAL to BIT.

CHARACTER

Character 0 becomes bit 0 and character 1 becomes bit 1. Any character other than 0 or 1 raises the CONVERSION condition. A null string becomes a null bit string. The generated bit string, which has the same length as the source char-

acter string, is assigned to the target according to the rules for string assignment.

GRAPHIC
Graphic variables and strings are converted to CHARACTER, and then follow the rules for character source described on page 83.

Target: GRAPHIC

Nongraphic source is first converted to character according to the rules in Target: Character on page 81. The resultant character string is then converted to a DBCS string.

Examples

DECIMAL FIXED to BINARY FIXED with fractions

```
dcl I fixed bin(31,5) init(1);
    I = I+.1;
```

The value of I is now 1.0625. This is because .1 is converted to FIXED BINARY(5,4), so that the nearest binary approximation is 0.0001B (no rounding occurs). The decimal equivalent of this is .0625. The result achieved by specifying .1000 in place of .1 would be different.

Arithmetic-to-bit-string conversion

```
DCL A bit(1),
    D bit(5);
A=1;          /* A has value '0'B     */
D=1;          /* D has value '00010'B */
D='1'B;       /* D has value '10000'B */
if A=1 then go to Y;
       else go to X;
```

The branch is to X, because the assignment to A resulted in the following sequence of actions:

1. The decimal constant, 1, has the attributes FIXED DECIMAL (1,0) and is assigned to temporary storage with the attributes FIXED BIN(4,0) and the value 0001B.
2. This value now converts to a bit string of length (4), so that it becomes '0001'B.
3. The bit string is assigned to A. Since A has a declared length of 1, and the value to be assigned has acquired a length of 4, truncation occurs at the right, and A has a final value of '0'B.

For the comparison operation in the IF statement, '0'B and 1 convert to FIXED BINARY and compared arithmetically. They are unequal, giving a result of *false* for the relationship A=1.

In the first assignment to D, a sequence of actions similar to that described for A takes place, except that the value is extended at the right with a zero, because D has a declared length that is 1 greater than that of the assigned value.

Arithmetic-value-to-character-string conversion

In the following example, the three blanks are necessary to allow for the possibility of a minus sign, a decimal or binary point, and provision for a single leading zero before the point:

```
dcl A char(4),
    B char(7);
A='0'; /*A has value '0bbb'*/
A=0;   /*A has value 'bbb0'*/
B=1234567; /*B has value 'bbb1234'*/
```

A conversion error

```
dcl Ctlno char(8) init('0');
do I=1 to 100;
    Ctlno=Ctlno+1;
    :
end;
```

For this example, FIXED DECIMAL precision 15 was used for the implementation maximum. The example raises the CONVERSION condition because of the following sequence of actions:

1. The initial value of CTLNO, that is, '0bbbbbbb' converts to FIXED DECIMAL(15,0).
2. The decimal constant, 1, with attributes FIXED DECIMAL(1,0), is added; in accordance with the rules for addition, the precision of the result is (16,0).
3. This value now converts to a character string of length 18 in preparation for the assignment back to CTLNO.
4. Because CTLNO has a length of 8, the assignment causes truncation at the right; thus, CTLNO has a final value that consists entirely of blanks. This value cannot be successfully converted to arithmetic type for the second iteration of the loop.

Chapter 6. Program Organization

Chapter 6. Program organization

This chapter discusses how statements can be organized into different kinds of blocks to form a PL/I program, how control flows among blocks, and how different blocks can make use of the same data.

Proper division of a program into blocks simplifies the writing and testing of the program, particularly when many programmers are writing it. Proper division can also result in more efficient use of storage, since automatic storage is allocated on entry to the block in which it is declared and released when the block is terminated.

Programs

Program structure

PL/I is a block-oriented language, consisting of packages, procedures, begin blocks, statements, expressions, and built-in functions.

A PL/I *application* consists of one or more separately loadable entities, known as a *load module*. An OS/2 load module has the file extension of EXE or DLL. Each load module may consist of one or more separately compiled entities, known as a *compilation unit (CU)*. Unless otherwise stated, a *program* refers to a PL/I application or a compilation unit.

A compilation unit is a PL/I PACKAGE or an external PROCEDURE. Each package may contain zero or more procedures, some or all of which may be exported. A PL/I external or internal procedure contains zero or more blocks. A PL/I block is either a PROCEDURE or a BEGIN block, which contains zero or more statements and/or zero or more blocks.

A PL/I block allows you to produce highly-modular applications, because blocks can contain declarations that define variable names and storage class. Thus, you can restrict the scope of a variable to a single block or a group of blocks, or can make it known throughout the compilation unit or a load module.

By giving you freedom to determine the degree to which a block is self-contained, PL/I makes it possible to produce blocks that many compilation units and applications can share, leading to code reuse.

Figure 24 on page 90 shows an application structure.

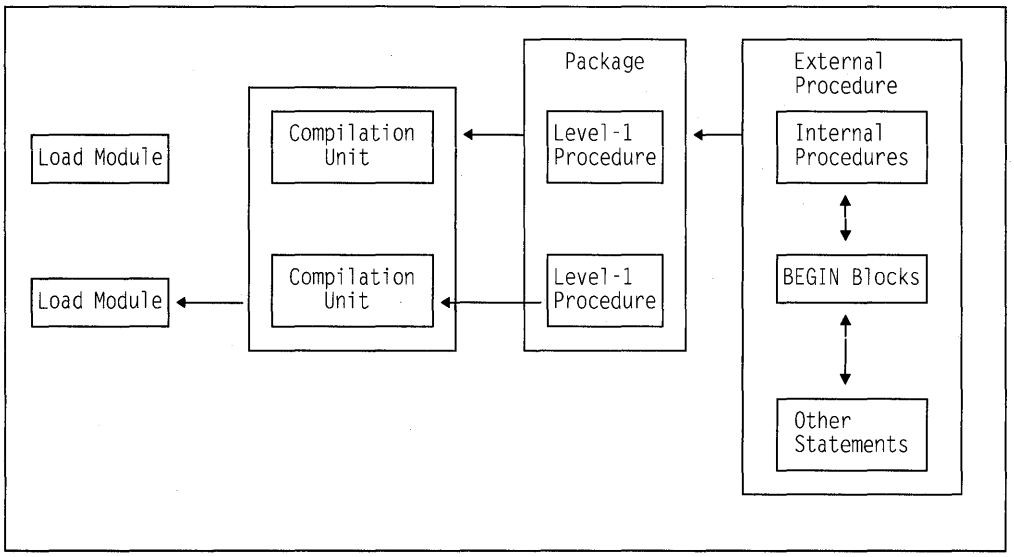


Figure 24. A PL/I application structure

Packages are discussed in "Packages" on page 92.

Procedures are discussed in "Procedures" on page 94.

Begin blocks are discussed in "Begin blocks" on page 111.

Program activation

A PL/I program becomes active when a calling program invokes the *main procedure*. This calling program usually is the operating system, although it could be another program. The main procedure is the external procedure for which the PROCEDURE statement has the OPTIONS(MAIN) specification. In the following example, `Contrl` is the main procedure and it invokes other external procedures in the program. The main procedure remains active for the duration of the program.

```
Contrl: procedure options(main);
  call A;
  call B;
  call C;
end contrl;
```

Program termination

A program is terminated when the main procedure is terminated. Whether termination is normal or abnormal, control returns to the calling program. In the previous example, when control transfers from the C procedure back to the `Contrl` procedure, `Contrl` terminates. See "Procedure termination" on page 99 for more information.

Blocks

A block is a delimited sequence of statements that does the following:

- Establishes the scope of names declared within it
- Limits the allocation of automatic variables
- Determines the scope of DEFAULT statements (as described in "Defaults for attributes" on page 141).

The kinds of blocks are:

- Package
- Procedure
- Begin.

These blocks can contain declarations that are treated as local definitions of names. This is done to establish the scope of the names and to limit the allocation of automatic variables. These declarations are not known outside their own block, and the names cannot be referred to in the containing block. See "Scope of declarations" for more information.

Storage is allocated to automatic variables upon entry to the block where the storage is declared, and is freed upon exit from the block. See "Scope of declarations," for more information.

Block activation

Each block plays the same role in the allocation and freeing of storage and in delimiting the scope of names. How activation occurs is discussed in "Procedures" on page 94 and "Begin blocks" on page 111. Packages are neither activated nor terminated.

During block activation, the following are performed:

- Expressions that appear in declare statements are evaluated for extents and initial values (including iteration factors).
- Storage is allocated for automatic variables. Their initial values are set if specified.
- Storage is allocated for dummy arguments and compiler-created temporaries that might be created in this block.

Initial values and extents for automatic variables must not depend on the values or extents of other automatic variables declared in the same block. For example, the following initialization can produce incorrect results for J and K:

```
dcl I init(10),J init(K),K init(I);
```

Declarations of data items must not be mutually interdependent. For example, the following declarations are invalid:

```
dcl A(B(1)), B(A(1));

dcl D(E(1)), E(F(1)), F(D(1));
```

Errors can occur during block activation, and the ERROR condition (or other conditions) can be raised. If so, the environment of the block might be incomplete. In

particular, some automatic variables might not have been allocated. Statements referencing automatic variables executed after the ERROR condition has been raised may reference unallocated storage. The results of referring to unallocated storage are undefined.

Block termination

There are a number of ways a block can be terminated. How termination occurs is discussed in "Procedures" on page 94 and "Begin blocks" on page 111. Packages are neither activated nor terminated.

During block termination:

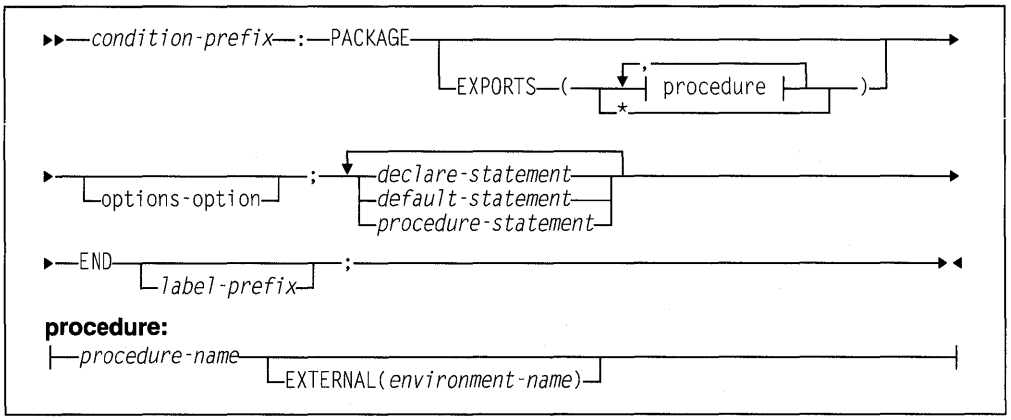
- The ON-unit environment is re-established as it existed before the block was activated.
- Storage for all automatic variables allocated in the block is released.

Packages

A package is a block that can only contain declarations, default statements, and procedure blocks. The package forms a name scope that is shared by all declarations and procedures contained in the package, unless the names are declared again. Some or all of the level 1 procedures can be exported and made known outside of the package as external procedures. A package can be used for implementing multiple entry point applications.

PACKAGE statement

The syntax for the PACKAGE statement is:



condition-prefix
Condition prefixes specified on a PACKAGE statement apply to all procedures contained in the package unless overridden on the PROCEDURE statement. For more information on condition prefixes, refer to "Condition prefixes" on page 302.

name
The name of the package.

EXPORTS
Specifies that all (EXPORTS(*)) or the named procedures are to be exported and thus made externally known outside of the package.

procedure name
Is the name of a level 1 procedure within the package.

EXTERNAL (environment name)
is a scope attribute discussed in "Scope of declarations" on page 132.

options option
For OPTIONS options applicable to a package statement, refer to "OPTIONS option and attribute" on page 121.

declare statement
All variables declared within a package but outside any contained level 1 procedure must have the storage class of static, based, or controlled. Automatic variables are not allowed. Default storage class is STATIC. Refer to Chapter 7, "Data declaration" on page 128.

default statement
Refer to "Defaults for attributes" on page 141.

procedure statement
Refer to "PROCEDURE statement" on page 94.

An example of the package statement appears in Figure 25.

```
*Process S A(F) LANGLVL(SAA2) LIMITS(EXTNAME(31)) NUMBER;
 Package_Demo: Package exports (Factorial);

 /*********************************************/
 /*                Common Data                */
 /*********************************************/

 Dcl n fixed bin(15);
 Dcl message char(*) value('The factorial of ');

 /*********************************************/
 /*                Main Program               */
 /*********************************************/

 Factorial: Proc Options (main);

    Dcl result fixed bin(31);

    put skip list('Please enter a number whose factorial ' ||
                  'must be computed ');
    get list(n);

    result = Compute_factorial(n);

    put list(message || trim(n) || ' is ' || trim(result));

 end Factorial;
```

Figure 25 (Part 1 of 2). Package statement

```
/*********************************************/
/*                Subroutine                 */
/*********************************************/

Compute_factorial: proc (input) recursive returns (fixed bin(31));

   dcl input fixed bin(15);

   if input <= 1 then
      return(1);
   else
      return( input*Compute_factorial(input-1) );

end Compute_factorial;

end Package_Demo;
```

Figure 25 (Part 2 of 2). Package statement

Procedures

A procedure is a sequence of statements delimited by a PROCEDURE statement and a corresponding END statement. A procedure can be a main procedure, a subroutine, or a function. An application must have exactly one external procedure that has OPTIONS(MAIN). In the following example, the name of the procedure is `Name` and represents the *entry point* of the procedure.

```
Name:
   procedure;
end Name;
```

A procedure must have a name. A procedure block nested within another procedure or begin block is called an *internal procedure*. A procedure block not nested within another procedure or begin block is called an *external procedure*. Level 1 exported procedures from a package also become external procedures. External procedures can be invoked by other procedures in other compilation units. Procedures can invoke other procedures.

A procedure can be recursive, which means that it can be reactivated from within itself or from within another active procedure while it is already active. You can pass arguments when invoking a procedure.

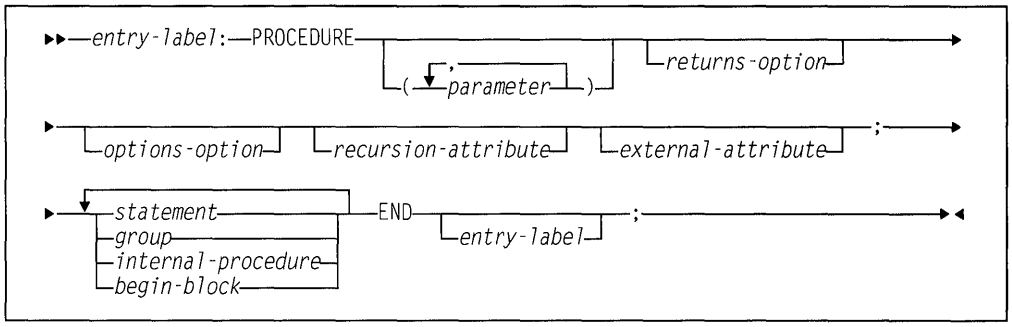
For more information on these subjects, see the following sections:

- "Scope of declarations" on page 132
- "Subroutines" on page 104
- "Functions" on page 106
- "Passing arguments to procedures" on page 108.

PROCEDURE statement

A procedure statement identifies the procedure as a main procedure, a subroutine, or a function. Parameters expected by the procedure and other characteristics are also specified on the procedure statement.

The syntax for the PROCEDURE statement is:



Abbreviations: PROC for PROCEDURE

entry-label
Is the entry point to the procedure. External entries are explicitly declared in the invoking procedure. Refer to "Entry data" on page 112 for more information.

parameter
Refer to "Parameter attribute" and "Passing arguments to procedures" on page 108.

returns-option
applies only to function procedures. Refer to "Functions" on page 106 and "RETURNS option and attribute" on page 126

options-option
Refer to "OPTIONS option and attribute" on page 121.

recursion-attribute
Refer to "Recursive procedures" on page 100.

external-attribute
Refer to "Scope of declarations" on page 132.

Parameter attribute

A parameter is contextually declared with the parameter attribute by its specification in the PROCEDURE statement. The parameter should be explicitly declared with appropriate attributes. The PARAMETER attribute may also be specified in the declaration. If attributes are not supplied in a DECLARE statement, default attributes are applied. The parameter name must not be subscripted or qualified.

The syntax of the parameter attribute is:

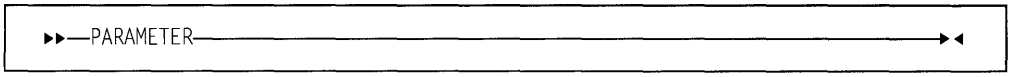


Figure 10 on page 29, and the following discussion, describe the attributes that can be declared for a parameter.

A parameter always has the INTERNAL attribute.

If the parameter is a structure or union, it must specify the level-1 name.

A parameter cannot have any storage class attributes except CONTROLLED. A controlled parameter must have a controlled argument, and can also have the INITIAL attribute.

Parameters used in record-oriented input/output, or as the base variable for DEFINED items, must be in connected storage. The CONNECTED attribute must be specified both in the declaration in the procedure and in the descriptor list of the procedure entry declaration.

Parameters and array arguments

If an argument is an array, a string, or an area, the bounds of the array, the length of the string, or the size of the area must be declared for the corresponding parameter. The number of dimensions and the bounds of an array parameter, or the size and length of an area or string parameter, must be the same as the current generation of the corresponding argument.

Noncontrolled parameter extents: Extents of noncontrolled parameters must be specified either by asterisks or by constants. When the actual extents could be different for different occasions, each can be specified in a DECLARE statement by an asterisk. When an asterisk is used, the extents are taken from the current generation of the associated argument.

Controlled parameter extents: The extents of a controlled parameter can be specified in a DECLARE statement either by asterisks or by element expressions.

Asterisk notation When asterisks are used, extents of the controlled parameter is taken from the current generation of the associated argument. Any subsequent allocation of the controlled parameter uses the same extents. Unless the argument in the calling procedure is a controlled parameter, it must be declared with nonrestricted expressions for its extents.

Expression notation
Each time the parameter is allocated, the expressions are evaluated to give current extents for the new allocation.

Example of array argument with parameters: In Figure 26 on page 97, when `Sub1` is invoked, `A` and `B`, which have been allocated, are passed.

```
%process or('|') num  margins(1,72);
Package:package exports(*);

  Main: procedure options(Main);
    declare (A(NA), B(NB), C(NC), D(ND) ) controlled;
    declare (NA init(20), NB init(30), NC init(100),
            ND init(100) ) fixed bin(31);
    declare Sub1 entry((*) controlled, (*) controlled);
    declare Sub2 entry ((*) ctl, (*) ctl, fixed bin);
    allocate A,B;  /* a(20), b(30) */
    display ('Gen1: DIM(A)=' || dim(a) || ', ' || "DIM(B)=" || dim(b));
    call Sub1(A,B);
    display ('Gen2: Allocn(A)=' || allocn(a) || ', ' ||
             'Allocn(B)=' || allocn(b) );
    display ('Gen2: DIM(A)=' || dim(a) || ', ' || "DIM(B)=" || dim(b));
    free A,B;
    display ('Gen1: Allocn(A)=' || allocn(a) || ', ' ||
             'Allocn(B)=' || allocn(b) );
    display ('Gen1: DIM(A)=' || dim(a) || ', ' || "DIM(B)=" || dim(b));
    free A,B;
    display ('Gen0: Allocn(A)=' || allocn(a) || ', ' ||
             'Allocn(B)=' || allocn(b) );
    call Sub2 (C,D,10);
    display ('Gen1: Allocn(C)=' || allocn(c) || ', ' ||
             'Allocn(D)=' || allocn(d) );
    display ('Gen1: DIM(C)=' || dim(c) || ', ' || "DIM(D)=" || dim(d));
    free C,D;
    display ('Gen0: Allocn(C)=' || allocn(c) || ', ' ||
             'Allocn(D)=' || allocn(d) );
  end Main;

  Sub1: procedure (U,V);
    dcl (U(UB), V(*)) controlled,
         UB fixed bin(31);
         display ('Gen1: Allocn(U)=' || allocn(u) || ', ' ||
                  'Allocn(V)=' || allocn(v) );
         display ('Gen1: DIM(U)=' || dim(u) || ', ' || "DIM(V)=" || dim(v));
         UB=200;
         allocate U,V; /* u(200), v(30) */
         display ('Gen2: Allocn(U)=' || allocn(u) || ', ' ||
                  'Allocn(V)=' || allocn(v) );
         display ('Gen2: DIM(U)=' || dim(u) || ', ' || "DIM(V)=" || dim(v));
    end Sub1;

  Sub2: procedure (X,Y,N);
    declare (X(N),Y(N)) controlled,
             N fixed bin;
                  display ('Gen0: Allocn(X)=' || allocn(x) || ', ' ||
                           'Allocn(Y)=' || allocn(y) );
                  allocate X,Y;  /* x(10), y(10) */
                  display ('Gen1: Allocn(X)=' || allocn(x) || ', ' ||
                           'Allocn(Y)=' || allocn(y) );
                  display ('Gen1: DIM(X)=' || dim(x) || ', ' || "DIM(Y)=" || dim(y));
                end Sub2;

              end Package;
```

Figure 26. Array argument with parameters

The ALLOCATE statement in Sub1 allocates a second generation of A and B. B has the same bounds for both generations, and A has different bounds for the second generation.

On return to Main, the first FREE statement frees the second generation of A and B (allocated in Sub1). The second FREE statement frees the first generation of A and B (allocated in Main).

In Sub2, X and Y are declared with bounds that depend on the value of N. When X and Y are allocated, their values determine the bounds of the allocated arrays.

On return to Main from Sub2, the FREE statement frees the only generation of C and D (allocated in Sub2).

Procedure activation

Sequential program flow passes around a procedure, from the statement before the PROCEDURE statement to the statement after the END statement of that procedure. The only way that a procedure can be activated is by a *procedure reference.* ("Program activation" on page 90 tells how to activate the main procedure.) The execution of the invoking procedure is suspended until the invoked procedure returns control to it.

A procedure reference is the appearance of an entry expression in one of the following contexts:

- Using a CALL statement to invoke a subroutine as described in "CALL statement."
- Invoking a function as described in "Functions."

The information in this section is relevant to each of these contexts. However, the examples in this chapter use CALL statements.

When a procedure reference occurs, the procedure containing the specified entry point is said to be *invoked.* The point at which the procedure reference appears is called the *point of invocation* and the block in which the reference is made is called the *invoking block.* An invoking block remains active even though control is transferred from it to the procedure it invokes.

When a procedure is invoked, arguments and parameters are associated and execution begins with the first statement in the invoked procedure.

This procedure:

```
Readin:  procedure;
  statement-1
  statement-2
  statement-3
  ⋮
end readin;
```

can be activated by this entry reference:

```
call Readin;
```

The entry constant (Readin) can also be assigned to an entry variable that is used in a procedure reference. For example:

```
declare Readin entry,
        Ent1 entry variable;
Ent1 = Readin;
call Ent1;
call Readin;
```

The two CALL statements have the same effect.

Procedure termination

A procedure is terminated when, by some means other than a procedure reference, control passes back to the invoking program, block, or to some other active block.

Procedures normally terminate when:

- Control reaches a RETURN statement within the procedure. The execution of a RETURN statement returns control to the point of invocation in the invoking procedure. If the point of invocation is a CALL statement, execution in the invoking procedure resumes with the statement following the CALL. If the point of invocation is a function reference, execution of the statement containing the reference is resumed.
- Control reaches the END statement of the procedure. Effectively, this is equivalent to the execution of a RETURN statement.

Procedures abnormally terminate when:

- Control reaches a GO TO statement that transfers control out of the procedure. The GO TO statement can specify a label in a containing block, or it can specify a parameter that has been associated with a label argument passed to the procedure.
- A STOP statement is executed.
- An EXIT statement is executed.
- The ERROR condition is raised and there is no established ON-unit for ERROR or FINISH. Also, if one or both of the conditions has an established ON-unit, ON-unit exit is by normal return, rather than by a GO TO statement.
- The procedure calls or invokes another procedure that terminates abnormally.

Transferring control out of a procedure using a GO TO statement can sometimes result in the termination of several procedures and/or begin-blocks. Specifically, if the transfer point specified by the GO TO statement is contained in a block that did not directly activate the block being terminated, all intervening blocks in the activation sequence are terminated.

In the following example:

```
A: procedure options(main);
   statement-1
   statement-2
   B: begin;
      statement-b1
      statement-b2
      call C;
      statement-b3
   end B;
   statement-3
   statement-4
   C: procedure;
      statement-c1
      statement-c2
      statement-c3
      D: begin;
         statement-d1
         statement-d2
         go to Lab;
         statement-d3
      end D;
      statement-c4
   end C;
   statement-5
Lab: statement-6
   statement-7
end A;
```

A activates B, which activates C, which activates D. In D, the statement `go to Lab` transfers control to `statement-6` in A. Since this statement is not contained in D, C, or B, all three blocks are terminated; A remains active. Thus, the transfer of control out of D results in the termination of intervening blocks B and C as well as the termination of block D.

Recursive procedures

An active procedure that is invoked from within itself or from within another active procedure is a *recursive* procedure. Such an invocation is called recursion.

A procedure that is invoked recursively must have the RECURSIVE attribute specified in the PROCEDURE statement. The syntax for the RECURSIVE attribute is shown in "PROCEDURE statement" on page 94.

The environment (that is, values of automatic variables, and so on) of every invocation of a recursive procedure is preserved in a manner analogous to the stacking of allocations of a controlled variable (see "Controlled storage and attribute" on page 201). Think of an environment as being *pushed down* at a recursive invocation, and *popped up* at the termination of that invocation. A label constant in the current block is always a reference to the current invocation of the block that contains the label.

If a label constant is assigned to a label variable in a particular invocation, and the label variable is not declared within the recursive procedure, a GO TO statement naming that variable in another invocation restores the environment that existed when the assignment was performed, terminating the current and any intervening procedures and begin blocks.

The environment of a procedure that was invoked from within a recursive procedure by means of an entry variable is the one that was current when the entry constant was assigned to the variable. Consider the following example:

```
I=1;
call A;                                 /* First invocation of A      */

A: proc recursive;
   declare Ev entry variable static;
   if I=1 then
      do;
        I=2;
        Ev=B;
        call A;                         /* 2nd invocation of A        */
      end;
   else call Ev;                        /* Invokes B with environment */
                                        /* of first invocation of A   */
B: proc;
   go to Out;
end B;
Out: end A;
```

The GO TO statement in the procedure B transfers control to the END A statement in the first invocation of A, and terminates B and both invocations of A.

Effect of recursion on automatic variables

The values of variables allocated in one activation of a recursive procedure must be protected from change by other activations. This is arranged by stacking the variables. A stack operates on a last-in, first-out basis. The most recent generation of an automatic variable is the only one that can be referenced. Static variables are not affected by recursion. Thus, they are useful for communication across recursive invocations. This also applies to automatic variables that are declared in a procedure that contains a recursive procedure and to controlled and based variables. In the following example:

```
A: proc;
   dcl X;
   :
   B: proc recursive;
       dcl Z,Y static;
       call B;
      :
    end B;
  end A;
```

A single generation of the variable X exists throughout invocations of procedure B. The variable Z has a different generation for each invocation of procedure B. The variable Y can be referred to only in procedure B and is not reallocated at each invocation. (The concept of stacking variables is also of importance in the discussion of controlled variables in "Controlled storage and attribute" on page 201)

Dynamic loading of an external procedure

A DLL can be dynamically fetched (loaded) or released (deleted) by a PL/I program using FETCH and RELEASE statements.

A procedure invoked by a procedure reference usually is resident in main storage throughout the execution of the program. However, a procedure can be loaded into main storage for only as long as it is required. The invoked procedure can be

dynamically loaded into, and dynamically deleted from, main storage during execution of the calling procedure.

Dynamic loading and deletion of procedures is particularly useful when a called procedure is not necessarily invoked every time the calling procedure is executed, and when conservation of main storage is more important than a short execution time.

The PL/I statements that initiate the loading and deletion of a procedure are FETCH and RELEASE statements respectively.

The appearance of an entry constant in a FETCH statement indicates that the referenced procedure needs to be loaded into main storage before it can be executed, unless a copy already exists in main storage. Provided the name is referenced in a FETCH statement, a procedure may also be loaded from the disk by:

- Execution of a CALL statement or the CALL option of an INITIAL attribute
- Execution of a function reference.

It is not necessary that control passes through a FETCH or RELEASE statement, either before or after execution of the CALL or function reference.

Whichever statement loaded the procedure, execution of the CALL statement or option or the function reference invokes the procedure in the normal way.

It is not an error if the procedure has already been loaded into main storage. The fetched procedure can remain in main storage until execution of the whole program is completed. Alternatively, the storage it occupies can be freed for other purposes at any time by means of the RELEASE statement.

Rules

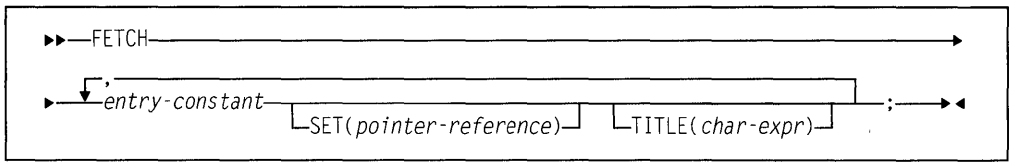
FETCH and RELEASE have the following restrictions:

- Only external procedures can be fetched.
- EXTERNAL CONDITION conditions are shared across the entire application (main and fetched modules). EXTERNAL files are shared in a limited way:
  - A FILE constant in an I/O statement (for example, OPEN, CLOSE READ, WRITE, PUT, or GET) is shared across the application.
  - Any FILE conditions (for example, ENDFILE), must be trapped in one linked unit.

  Other external variables are shared only within a single module.
- A dynamic link library (DLL) contains one or more segments. Each segment contains one or more fetched procedures. A segment is loaded into memory when a procedure is fetched from a segment that is not already in memory. The memory will not be freed until all of the fetched procedures within a DLL have been released.
- Storage for STATIC variables in the fetched procedure is allocated when the segment containing the procedure is loaded into memory. Each time a segment is loaded into memory, the STATIC variables are given the initial values indicated by their declarations.
- The FETCH and RELEASE statements must specify entry constants. An entry constant for a fetched procedure may be assigned to an entry variable provided the procedure has been fetched.

FETCH statement

The FETCH statement checks main storage for the named procedures. Procedures not already in main storage are loaded from the disk. COBOL, FORTRAN, or C routines cannot be fetched. The syntax for the FETCH statement is:



entry-constant

specifies the name by which the procedure to be fetched is known to the operating system. Details of the linking considerations for fetchable procedures are given in the *PL/I Package/2 Programming Guide*.

The entry-constant must be the same as the one used in the corresponding CALL statement, CALL option, or function reference.

SET SET specifies a pointer reference that will be set to the address of the entry point of the loaded module. This option can be used to load tables (non-executable load modules). It can also be used for entries that are fetched and whose address needs to be passed to non-PL/I procedures.

If the load module is later released by the RELEASE statement, and the load module is accessed (through the pointer), unpredictable results can occur.

TITLE If TITLE is specified, the load module name specified is searched for and loaded. If it is not specified, the load module name used is the environment name specified in the EXTERNAL attribute for the variable (if present) or the entry constant name itself. For example:

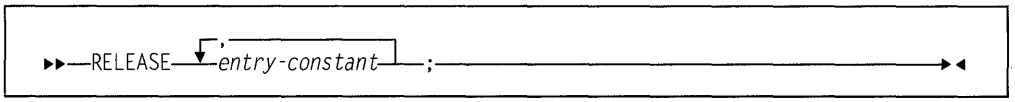
```
dcl a entry;
dcl b entry ext('C');

fetch a title('X');          /* X is loaded       */
fetch a;                     /* A is loaded       */
fetch b title('Y');          /* Y is loaded       */
fetch b;                     /* C is loaded       */
```

For information on the types of title strings, refer to *PL/I Package/2 Programming Guide*.

RELEASE statement

The RELEASE statement frees the main storage occupied by procedures identified by its specified entry-constants.

The syntax for the RELEASE statement is:



entry-constant

must be the same as the one used in the corresponding CALL statement, CALL option, or function reference, and FETCH statements.

Consider the following example, in which ProgA and ProgB are entry names of procedures resident on disk:

```
Prog:  procedure;

[1]      fetch ProgA;
[2]      call ProgA;
[3]      release ProgA;

[4]      call ProgB;
         go to Fin;

         fetch ProgB;
Fin:  end Prog;
```

[1] ProgA is loaded into main storage by the first FETCH statement

[2] ProgA executes when the first CALL statement is reached.

[3] Storage for ProgA is released when the RELEASE statement is executed.

[4] ProgB is loaded and executed when the second CALL statement is reached, even though the FETCH statement referring to this procedure is never executed.

The same results would be achieved if the statement FETCH ProgA were omitted. The appearance of ProgA in a RELEASE statement causes the statement CALL ProgA to load the procedure, as well as invoke it.

The fetched procedure is compiled and linked separately from the calling procedure. You must ensure that the entry constant specified in FETCH, RELEASE, and CALL statements and options, and in function references, is the name known on the disk. This is discussed in your *PL/I Package/2 Programming Guide*.

Subroutines

A *subroutine* is an internal or external procedure that is invoked by a CALL statement. The syntax for a subroutine is in "Procedures" on page 94.

The arguments of the CALL statement are associated with the parameters of the invoked procedure. The subroutine is activated, and execution begins. The arguments (zero or more) can be input only, output only, or both.

A subroutine is normally terminated by the RETURN or the END statement. Control is then returned to the invoking block. A subroutine can be abnormally terminated as described in "Procedure termination" on page 99.

A subroutine procedure must:

- Not have the RETURNS option on the procedure statement.
- Not be declared as an entry with the RETURNS attribute if it is an external procedure.
- Be invoked using the CALL statement, not a function reference.
- Not return a result value using the RETURN statement.

The following examples illustrate the invocation of subroutines that are internal to and external to the invoking block.

Example 1

```
Prmain:  procedure;
         declare Name character (20),
         Item bit(5),
|4|      Outsub entry;
|1|      call outsub (Name, Item);
end Prmain;

|2| Outsub:  procedure (A,B);
         declare A character (20),
         B bit(5);
|3|      put list (A,B);
end Outsub;
```

|1| The CALL statement in Prmain invokes the procedure Outsub in |2| with the arguments Name and Item.

|2| Outsub associates Name and Item passed from Prmain with its parameters, A and B. When Outsub is executed, each reference to A is treated as a reference to Name. Each reference to B is treated as a reference to Item.

|3| The put list (A,B) statement transmits the values of Name and Item to the default output file, SYSPRINT.

|4| In the declaration of Outsub as an entry constant, no parameter descriptor has to be given with the ENTRY attribute, because the attributes of the arguments and parameters match. Also see "ENTRY attribute" on page 114.

Example 2

```
A:  procedure;
    declare Rate float (10),
            Time float(5),
            Distance float(15),
            Master file;
|1|   call Readcm (Rate, Time, Distance, Master);

|3|  Readcm:
|2|    procedure (W,X,Y,Z);
      declare W float (10),
              X float(5),
              Y float(15), Z file;
      get File (Z) list (W,X,Y);
      Y = W*X;
      if Y > 0 then
        return;
      else
        put list('ERROR READCM');
      end Readcm;

    end A;
```

|1| The arguments Rate, Time, Distance, and Master are passed to the procedure Readcm in |3| and associated with the parameters W, X, Y, and Z.

|2| A reference to W is the same as a reference to Rate, X the same as Time, Y the same as : Distance, and Z the same as Master.

|3| Note that Readcm is not explicitly declared in A. It is implicitly declared with the ENTRY attribute by its specification on the PROCEDURE statement.

Built-in subroutines

You can use *built-in subroutines*, which provide ready-made programming tasks. Their *built-in names* can be explicitly declared with the BUILTIN attribute. (For more information on the BUILTIN attribute refer to "Declaring built-in functions" on page 371.)

Each built-in subroutine is described in Chapter 17, "Built-in functions, pseudovariables, and subroutines."

Functions

A *function* is a procedure that has zero or more arguments and is invoked by a *function reference* in an expression. The function reference transfers control to a function procedure, and returns a value, and control, to replace the function reference in the evaluation of the expression. Aggregates cannot be returned, and ENTRY variables cannot be returned unless they have the LIMITED attribute. The evaluation of the expression then continues.

A function procedure must:

- Have the RETURNS option on the procedure statement.
- Be declared as an entry with the RETURNS attribute if it is an external procedure.
- Be invoked using a function reference. The CALL statement can be used to invoke it only if the returned value has the OPTIONAL attribute. In this case, the returned value is discarded upon return.
- Have matching attributes in the RETURNS option and in the RETURNS attribute.
- Use the RETURN statement to return control and the result value.

Whenever a function is invoked, the arguments in the invoking expression are associated with the parameters of the entry point. Control is then passed to that entry point. The function is activated and execution begins.

The RETURN statement terminates a function and returns the value specified in its expression to the invoking expression. See "RETURN statement" on page 121 for more information.

A function can be abnormally terminated as described in "Procedure termination" on page 99. If this method is used, evaluation of the expression that invoked the function is not completed, and control goes to the designated statement.

In some instances, a function can be defined so that it does not require an argument list. In such cases, the appearance of an external function name within an expression is recognized as a function reference only if the function name has been explicitly declared as an entry name. See "Entry invocation or entry value" on page 120 for additional information.

Examples

The following examples illustrate the invocation of functions that are internal to and external to the invoking block.

Example 1: In the following example, the assignment statement contains a reference to the Sprod function:

```
     Mainp: procedure;
            get list (A, B, C, Y);
|1|         X = Y**3+Sprod(A,B,C);

|2|  Sprod: procedure (U,V,W)
            returns (bin float(21));
            dcl (U,V,W) bin float(53);
            if U > V + W then
|3|           return (0);
            else
|3|           return (U*V*W);
      end Sprod;
```

|1| When Sprod is invoked, the arguments A, B, and C are associated with the parameters U, V, and W in |2|, respectively.

|2| Sprod is a function because RETURNS appears in the procedure statement. It is internal, and therefore needs no explicit entry declaration. If Sprod were external, Mainp would contain an entry declaration with RETURNS specified.

|3| Sprod returns either 0 or the value represented by U*V*W, along with control to the expression in Mainp. The returned value is taken as the value of the function reference, and evaluation of the expression continues.

Example 2

```
Mainp:  procedure;
        dcl Tprod entry (bin float(53),
                         bin float(53),
                         bin float(53),
                         label) external
          returns (bin float(21));
        get list (A,B,C,Y);
|1|      X = Y**3+Tprod(A,B,C,Lab1);
Lab1:  call Errt;
end Mainp;

|1|  Tprod:  procedure (U,V,W,Z)
            returns (bin float(21));
            dcl (U,V,W) bin float(53);
            declare Z label;

|2|          if U > V + W then
                go to Z;
|3|          else
                return (U*V*W);
      end Tprod;
```

|1| When Tprod is invoked, Lab1 is associated with parameter Z.

|2| If U is greater than V + W, control returns to Mainp at the statement labeled Lab1. Evaluation of the assignment in |1| is discontinued.

|3| If U is not greater than V + W, U*V*W is calculated and returned to Mainp in the normal fashion. Evaluation of the assignment in |1| continues.

Notice that `Tprod` is an external procedure. It has an explicit entry declaration in `Mainp`, which contains RETURNS.

Built-in functions

Besides allowing programmer-written function procedures, PL/I provides a set of *built-in functions*. Built-in functions include commonly used arithmetic functions and others, such as functions for manipulating strings and arrays. Built-in functions are invoked in the same way that you invoke programmer-defined functions. However, many built-in functions can return an array of values, whereas a programmer-defined function can return only an element value. The built-in names for built-in functions can be explicitly declared with the BUILTIN attribute. (For more information on the BUILTIN attribute, refer to "Declaring built-in functions" on page 371.)

Each built-in function is described in Chapter 17, "Built-in functions, pseudovariables, and subroutines."

Passing arguments to procedures

When a function or a subroutine is invoked, parameters are associated, from left to right, with the passed arguments. The number of arguments and parameters must be the same.

In general:

- Computational data arguments can be passed to parameters of any computational data type.
- Program control data arguments must be passed to parameters of the same type, with these exceptions.
  - Pointer and offset can be passed to each other.
  - LIMITED ENTRY can be passed to ENTRY, but ENTRY cannot be passed to LIMITED ENTRY.
  - An array of label constants cannot be used as an argument.

Arguments that require aggregate temporaries are not allowed. For example, the following statements are invalid as arguments if `a`, `b`, or `c` is an array.

```
call X( cos(a) )

call X( cos(b+c) )
```

For example, instead of coding:

```
dcl A(10) float bin(53), X entry;
call X(cos(A));
```

you could code:

```
dcl temp(hbound(A,1)) float bin(53);
temp = cos(A);
call X(temp);
```

Expressions in the argument list are evaluated in the invoking block before the subroutine or function is invoked. A parameter has no storage associated with it. It is merely a means of allowing the invoked procedure to access storage allocated in the invoking procedure.

Using BYVALUE and BYADDR

Unless an argument is passed BYVALUE, a reference to an argument, not its value, is generally passed to a subroutine or function. This is known as passing arguments by reference or BYADDR. A reference to a parameter in a procedure is a reference to the corresponding argument. Any change to the value of a parameter is actually a change to the value of the corresponding argument. However, this is not always possible or desirable. Constants, for example, should not be altered by an invoked procedure. For arguments that should not change, a *dummy argument* containing the value of the original argument is passed. Any reference to the parameter then is a reference to the dummy argument and not to the original argument.

Dummy arguments

A dummy argument is created when the argument is any of the following:

- A constant.
- An expression with operators, parentheses, or function references.
- A variable whose data attributes or alignment attributes or connected attribute are different from the attributes declared for the parameter.

  This does not apply to noncontrolled parameters when only bounds, lengths, or size differ and these are declared with asterisks

  This does not apply when an expression other than a constant is used to define the extents of a controlled parameter. In this case, argument and parameter extents are assumed to match.

  In the case of arguments and parameters with the PICTURE attribute, a dummy argument is created unless the picture specifications match exactly, after any repetition factors are applied. The only exception is that an argument or parameter with a + sign in a scaling factor matches a parameter or argument without the + sign.
- A controlled string or area (because an ALLOCATE statement could change the length or extent).
- A string or area with an adjustable length or size, associated with a noncontrolled parameter whose length or size is a constant.

Deriving dummy argument attributes

PL/I derives the attributes of a dummy arguments as follows:

- From the attributes declared for the associated parameter in an internal procedure.
- From the attributes specified in the parameter descriptor for the associated parameter in the declaration of the external entry. If there was not a descriptor for this parameter, the attributes of the constant or expression are used.
- From the extents (when specified by an asterisk in a declaration) of the argument for the bounds of an array, the length of a string, or the size of an area.

Rules for dummy arguments

The following rules apply to dummy arguments:

- If a parameter is an element (that is, a variable that is neither a structure nor an array) the argument must be an element expression.
- When a VARYING string element is passed to a NONVARYING parameter, whose length is undefined (that is, specified by an asterisk), a dummy argument with the current length of the original
- Entry variables passed as arguments are assumed to be aligned, so that no dummy argument is created when only the alignments of argument and parameter differ. See "Generic entries" on page 118, for a description of generic name arguments for entry parameters.
- If the parameter is of the program control data type (except locator), the argument must be a reference of the same data type.
- If a parameter is a locator (pointer or offset), the argument must be a locator. If the types differ, a dummy argument is created. The parameter descriptor of an offset parameter must not specify an associated area.
- A noncontrolled parameter can be associated with an argument of any storage class. However, if more than one generation of the argument exists, the parameter is associated only with that generation existing at the time of invocation.
- If the parameter is controlled, you must explicitly state this in the parameter descriptor for the ENTRY declaration.

  In addition, a controlled parameter must always have a corresponding controlled argument that cannot be subscripted, cannot be an element of a structure, and cannot cause a dummy to be created. If more than one generation of the argument exists at the time of invocation, the parameter corresponds to the entire stack of generations in existence. Consequently, at the time of invocation, a controlled parameter represents the current generation of the corresponding argument. A controlled parameter can be allocated and freed in the invoked procedure, allowing the manipulation of the allocation stack of the associated argument.

  If the extents of the controlled parameter are specified as asterisks or nonrestricted expressions, the original declaration must have extents declared as nonrestricted expressions.

Passing arguments to the MAIN procedure

The PROCEDURE statement for the main procedure can have a parameter list. Such parameters require no special considerations in PL/I. However, you must be aware of any requirements of the invoking program (for example, not to use a parameter as the target of an assignment).

When the invoking program is the operating system, a single argument is passed to the program. If this facility is used, the parameter must be declared as a VARYING character string within the procedure. The current length is set equal to the argument length at run-time. In the following example:

```
Tom:  proc (Param) options (main);
   dcl Param char(100) varying;
```

storage is allocated only for the current length of the argument.

When the MAIN and NOEXECOPS options are specified, the main procedure can have a single parameter that is a VARYING CHARACTER string. The parameter passes as is, and a descriptor is set up. (The slash (/) symbol, if contained in the string, is treated as part of the string). For example:

```
main:  proc(Parm) options(main noexecops);
   dcl Parm char(n) varying;
```

Begin blocks

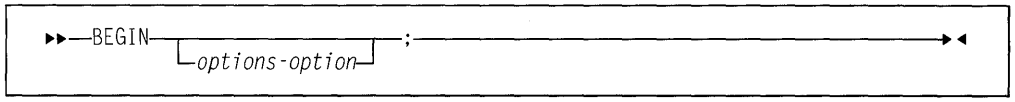
A begin block is a sequence of statements delimited by a BEGIN statement and a corresponding END statement. For example:

```
B:  begin;
  statement-1
  statement-2
  :
  statement-n
end B;
```

BEGIN statement

The BEGIN statement and a corresponding END statement delimit a begin block.

The syntax for the BEGIN statement is:



options-option
For the BEGIN block options, refer to "OPTIONS option and attribute" on page 121.

Begin block activation

Begin blocks are activated through sequential flow or as a *unit* in an IF, ON, WHEN, or OTHERWISE statement.

You can transfer control to a labeled BEGIN statement by issuing the GO TO statement.

Begin block termination

A begin block is terminated when control passes to another active block by some means other than a procedure reference. The methods are:

- The END statement for the begin block is executed. Control continues with the statement physically following the END, except when the block is an ON-unit.
- A GO TO statement within the begin block (or within any block internal to it) is executed. Control transfers to the point outside the block.
- A STOP or an EXIT statement is executed.
- Control reaches a RETURN statement that transfers control out of the begin block and out of its containing procedure. A RETURN statement in a begin block cannot contain an expression.

A GO TO statement can also terminate other blocks if the transfer point is contained in a block that did not directly activate the block being terminated. In this case, all intervening blocks in the activation sequence are terminated. For an example of this, see the example in "Procedure termination" on page 99.

Entry data

The entry data can be an entry constant or the value of an entry variable.

An entry constant is a name prefixed to a PROCEDURE statement, or a name declared with the ENTRY attribute and not the VARIABLE attribute. It can be assigned to an entry variable. In the following example, P, E1, and E2 are entry constants. Ev is an entry variable.

```
P:  procedure;
    declare Ev entry variable,
     (E1,E2) entry;

Ev = E1;
call Ev;
Ev = E2;
call Ev;
```

The first CALL statement invokes the entry point E1. The second CALL invokes the entry point E2.

The following example declares F(5), a subscripted entry variable.

The five entries A, B, C, D, and E are each invoked with the parameters X, Y, and Z.

```
declare (A,B,C,D,E) entry,
declare F(5) entry variable initial (A,B,C,D,E);
do I = 1 to 5;
  call F(I) (X,Y,Z);
end;
```

When an entry constant that is an entry point of an internal procedure is assigned to an entry variable, the assigned value remains valid only as long as the block that the entry constant was internal to remains active (and, for recursive procedures, remains current).

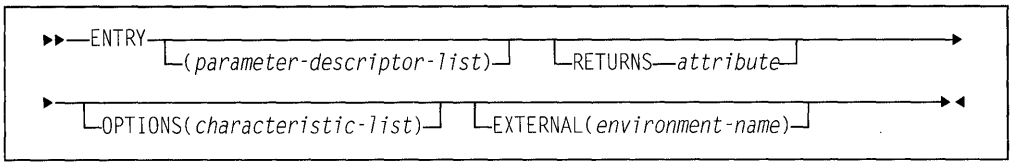
Entry constants

The appearance of a label prefix to a PROCEDURE statement explicitly declares an entry constant. A parameter-descriptor list is obtained from the parameter declarations, if any, and by defaults.

External entry constants must be explicitly declared. This declaration:

- Defines an entry point to an external procedure.
- Optionally specifies a parameter-descriptor list (the number of parameters and their attributes), if any, for the entry point.
- Specifies the attributes of the value that is returned by the procedure if the entry is a function.

The syntax for an entry constant is:



The attributes can appear in any order.

ENTRY attribute
For complete ENTRY attribute syntax, refer to "ENTRY attribute" on page 114.

OPTIONS attribute
For complete OPTIONS attribute syntax, refer to "OPTIONS option and attribute" on page 121.

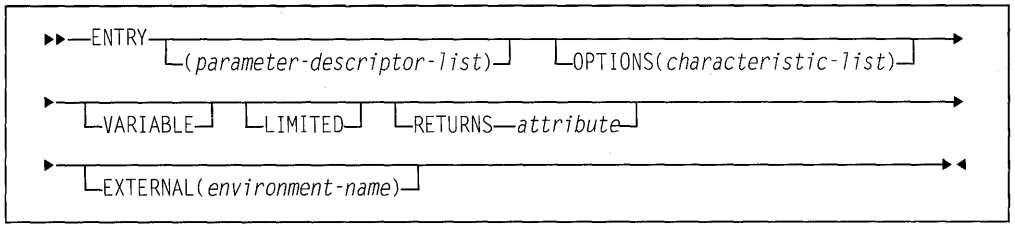
RETURNS attribute
For complete RETURNS attribute syntax, refer to "RETURNS option and attribute" on page 126.

EXTERNAL attribute
For a complete description of the EXTERNAL attribute refer to "INTERNAL and EXTERNAL attributes" on page 134.

Entry variables

An entry variable can contain both internal and external entry values. It can be part of an aggregate. For structuring and array dimension attributes, refer to "Arrays" on page 144 and "Structures" on page 147.

The syntax for an entry variable is:



The options can appear in any order.

ENTRY attribute
Refer to "ENTRY attribute" on page 114.

OPTIONS attribute
Refer to "OPTIONS option and attribute" on page 121.

VARIABLE attribute
The VARIABLE attribute establishes the name as an entry variable. This variable can contain entry constants and variables. Refer to "VARIABLE attribute" on page 48 for syntax information.

LIMITED attribute
Refer to "LIMITED attribute" on page 117.

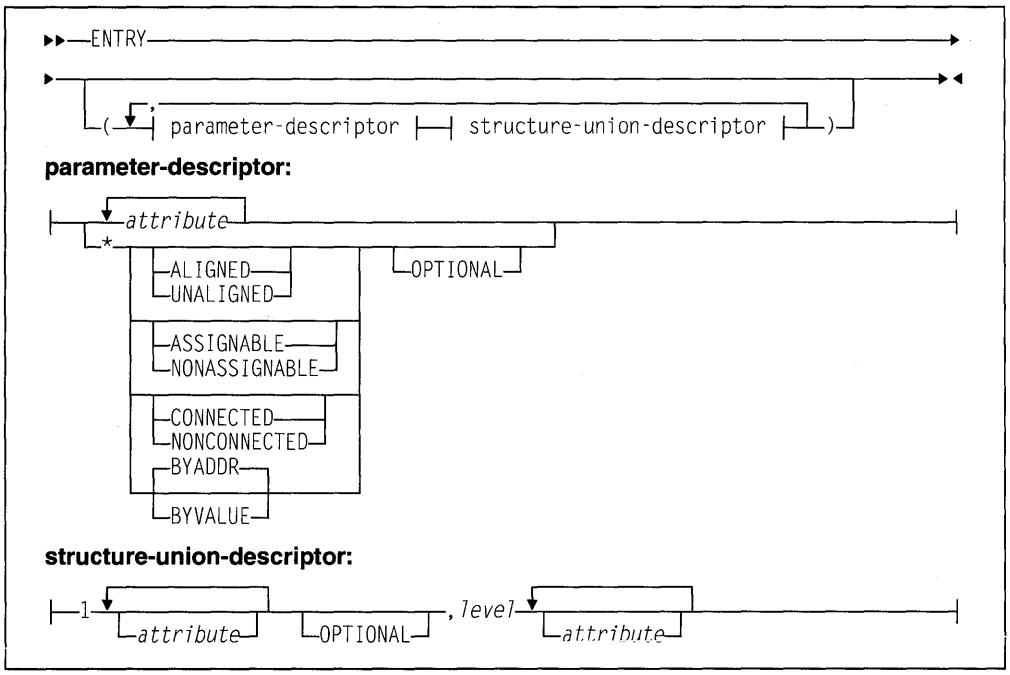
RETURNS attribute
Refer to "RETURNS option and attribute" on page 126.

EXTERNAL attribute
Refer to "Scope of declarations" on page 132.

ENTRY attribute

The ENTRY attribute specifies that the name being declared is either an external entry constant, or an entry variable. It also describes the attributes of the parameters of the entry point.

The syntax for the ENTRY attribute is:



ENTRY The ENTRY attribute, without a parameter descriptor list, is implied by the RETURNS attribute.

parameter-descriptor
A parameter descriptor list can be given to describe the attributes of the parameters of the associated external entry constant or entry variable. It is used for argument and parameter attribute matching and the creation of dummy arguments.

If no parameter descriptor list is given, the default is for the argument attributes to match the parameter attributes. Thus, the parameter descriptor list must be supplied if argument attributes do not match the parameter attributes.

Each parameter descriptor corresponds to one parameter of the entry point invoked and, if given, specifies the attributes of that parameter.

The parameter descriptors must appear in the same order as the parameters they describe. If a descriptor is absent, the default is for the argument to match the parameter.

If a descriptor for a parameter is not required, the absence of the descriptor must be indicated by an asterisk. For example:

`entry(character(10),*,*,fixed dec)`	Indicates four arguments.
`entry(*)`	Indicates one argument.
`entry( )`	Specifies that the entry name must never have any arguments.
`entry`	Specifies that it can have any number of arguments.
`entry(float binary,*)`	Indicates two arguments.

attribute

The attributes can appear in any order in a parameter descriptor. For an array parameter-descriptor, the dimension attribute must be the first specified.

*****

An asterisk specifies that, for that parameter, any data type is allowed. The valid attributes following the asterisk are:

- ALIGNED or UNALIGNED
- ASSIGNABLE or NONASSIGNABLE
- BYADDR or BYVALUE
- CONNECTED or NONCONNECTED
- OPTIONAL

No conversions will be done.

OPTIONAL

is discussed in "OPTIONAL attribute" on page 116.

structure-union-descriptor

For a structure union descriptor, the descriptor level numbers need not be the same as those of the parameter, but the structuring must be identical. The attributes for a particular level can appear in any order.

Defaults are not applied if an asterisk is specified. For example, in the following declaration defaults are applied only for the second parameter.

```
dcl X entry(* optional, aligned); /* defaults applied for 2nd parm */
```

Extents (lengths, sizes, and bounds) in parameter descriptors must be specified as constants or as asterisks. Controlled parameters must have asterisks.

RETURNS attribute implies the ENTRY attribute. For example:

Example parameter descriptors

```
Test:  procedure (A,B,C,D,E,F);

 declare A fixed decimal (5),
         B float binary (21),
         C pointer,
         1 D,
           2 P,
           2 Q,
             3 R fixed decimal,
         1 E,
           2 X,
           2 Y,
             3 Z,
         F(4) character (10);
 end Test;
```

Declarations for example descriptors

```
declare Test entry
            (decimal fixed (5),
             binary float (21),
           *,
            1,
             2,
             2,
              3 decimal fixed,
            *,
           (4) char(10));
```

In the previous example, the parameter C, and the structure parameter E do not have descriptors.

OPTIONAL attribute

OPTIONAL can be specified as part of the parameter-descriptor list or as an attribute in the parameter declaration.

OPTIONAL arguments can be omitted in calls and function references by specifying an asterisk for the argument. An omitted item can be anywhere in the argument list, including at the end. However, the omitted item is counted as an argument. With its inclusion in an entry, the number of arguments must not exceed the maximum number allowed for the entry.

Using OPTIONAL and BYVALUE for the same item is invalid.

The receiving procedure can use the OMITTED built-in function to determine if an OPTIONAL parameter/argument was omitted in the invocation of the entry. (For more information on the OMITTED built-in function, refer to "OMITTED" on page 418.)

Figure 27 shows both valid and invalid CALL statements for the procedure Vrtn. Vrtn determines if OPTIONAL parameters were omitted, and takes the appropriate action.

```
Caller: proc;
  dcl Vrtn entry (
           fixed bin,
           ptr optional,
           float,
           * optional);

/*  The following calls are valid:  */

  call Vrtn(10, *, 15.5, 'abcd');
  call Vrtn(10, *, 15.5, *);
  call Vrtn(10, addr(x), 15.5, *);

/*  The following calls are invalid:  */

  call Vrtn(10, addr(x), *, 'display');
  call Vrtn(10, addr(x), 15.5);
  call Vrtn(*, addr(x));
  call Vrtn(10,addr(x));
  call Vrtn(10);
  call Vrtn;
end Caller;

Vrtn: proc (Fb, P, Fl, C1);
  dcl Fb fixed bin,
      P ptr optional,
      Fl float,
      C1 char(8) optional;

  if ¬omitted(C1) then display (C1);
  if ¬omitted(P) then P=P+10;
end;
```

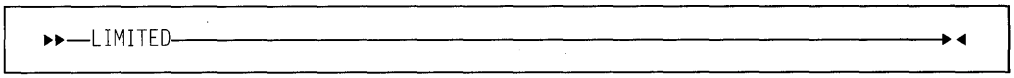
Figure 27. Valid and invalid call statements

LIMITED attribute

The LIMITED attribute indicates that the entry variable will have only non-nested entry constants as values. A entry variable that is not LIMITED can have entry constants as values.

A LIMITED static entry variable may be initialized with the value of a non-nested entry constant, thus permitting generation of more efficient code. It also uses less storage than a non-LIMITED entry variable.

The syntax for the LIMITED attribute is:



Example:

```
Example: proc options(reorder reentrant);
  dcl (Read, Write) entry;
  dcl FuncRtn(2) entry limited
       static init (Read, Write);

  dcl (prt1) entry;
  dcl PrtRtn(2) entry variable limited
       static init (Prt1,       /*  legal    */
                    Prt2);      /*  illegal  */
  Prt2: proc;
  :
  end Prt2;
end Example;
```

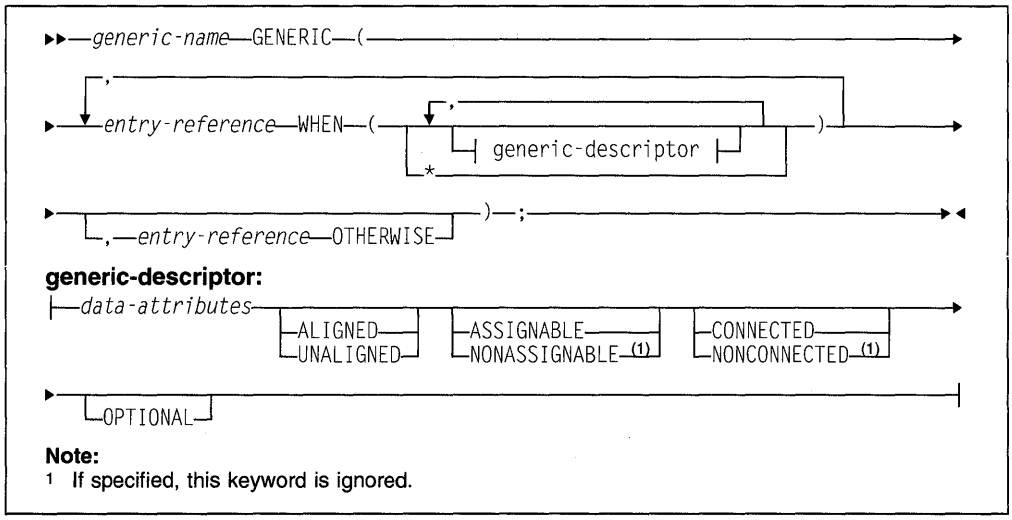
Generic entries

A generic entry declaration specifies a generic name for a set of entry references and their descriptors. During compilation, invocation of the generic name is replaced by one of the entries in the set.

GENERIC attribute

The generic name must be explicitly declared with the GENERIC attribute.

The syntax for declaring generic entry data is:



Abbreviation: OTHER for OTHERWISE

For the general declaration syntax, see page "DECLARE statement" on page 129.

entry-reference
must not be subscripted or defined. The same entry-reference can appear more than once within a single GENERIC declaration with different lists of descriptors.

generic-descriptor
corresponds to a single argument. It specifies an attribute that the corresponding argument must have so that the associated entry reference can be selected for replacement.

Structures or unions cannot be specified.

Where a descriptor is not required, its absence must be indicated by an asterisk.

The descriptor that represents the absence of all arguments in the invoking statement is expressed by omitting the generic descriptor in the WHEN clause of the entry. It has the form:

```
generic (... entry1 when( ) ...)
```

data-attributes
The data attributes are listed in "Data types and attributes" on page 25.

ALIGNED and UNALIGNED
are discussed in "ALIGNED and UNALIGNED attributes" on page 139.

ASSIGNABLE and NONASSIGNABLE
are discussed in "ASSIGNABLE and NONASSIGNABLE attributes" on page 216.

CONNECTED and NONCONNECTED
are discussed in "CONNECTED and NONCONNECTED attributes" on page 217.

OPTIONAL
is discussed in "OPTIONAL attribute" on page 116.

When an invocation of a generic name is encountered, the number of arguments specified in the invocation and their attributes are compared with descriptor list of each entry in the set. The first entry reference for which the descriptor list matches the arguments both in number and attributes replaces the generic name.

In the following example, an entry reference that has exactly two descriptors with the attributes DECIMAL or FLOAT, and BINARY or FIXED is searched for.

```
declare Calc generic (
             Fxdcal when (fixed,fixed),
             Flocal when (float,float),
             Mixed  when (float,fixed),
             Error otherwise);
Dcl      X decimal float (6),
         Y binary fixed (15,0);

         Z = X+Calc(X,Y);
```

If an entry with the exact number of descriptors with the exact attributes is not found, the entry with the OTHERWISE clause is selected if present. In the previous example, `Mixed` is selected as the replacement.

In a similar manner, an entry can be selected based on the dimensionality of the arguments.

```
dcl  D generic (D1 when ((*))),
                D2 when((*,*))),
     A(2),
     B(3,5);
call D(A);       /* D1 selected because A has one dimension  */
call D(B);       /* D2 selected because B has two dimensions */
```

If all of the descriptors are omitted or consist of an asterisk, the first entry reference with the correct number of descriptors is selected.

An entry expression used as an argument in a reference to a generic value only matches a descriptor of type ENTRY. If there is no such description, the program is in error.

Entry invocation or entry value

There are times when it may not be apparent whether an entry value itself will be used or the value returned by the entry invocation will be used. The following table and example help you understand which happens when.

If the entry reference . . .	It is . . .
Is a built-in function	Invoked
Has an argument list, even if null	Invoked
Is referenced in a CALL statement	Invoked
Has no argument list and is not referenced in a CALL statement	Not Invoked

In the following example, `A` is invoked, `B(C)` passes `C` as an entry value, and `D( C() )` invokes `C`.

```
dcl ( A, B, C returns (fixed bin), D) entry;

call A;                         /* A is invoked                    */
call B(C);                      /* C is passed as an entry value   */
call D( C() );                  /* C is invoked                    */
```

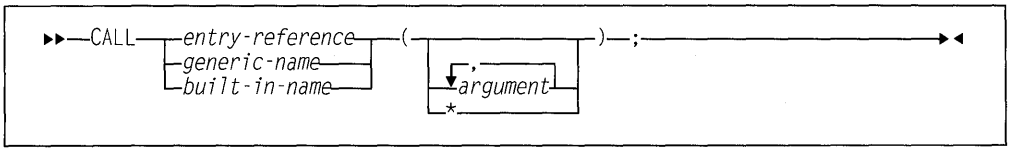
In the following example, the first assignment is invalid because it represents an attempt to assign an entry constant to an integer. The second assignment is valid.

```
dcl A fixed bin,
    B entry returns ( fixed bin );

A = B;
A = B();
```

CALL statement

The CALL statement invokes a subroutine.

The syntax for the CALL statement is:



entry-reference
specifies that the name of the subroutine to be invoked is declared with the ENTRY attribute (discussed in "Entry data" on page 112).

generic-name
specifies that the name of the subroutine to be invoked is declared with the GENERIC attribute (discussed in "Generic entries" on page 118).

built-in name
specifies the name of the subroutine to be invoked is declared with the BUILTIN attribute, (discussed in "Declaring built-in functions" on page 371).

argument
Element or an element expression or an aggregate to be passed to the invoked subroutine. See "Passing arguments to procedures" on page 108.

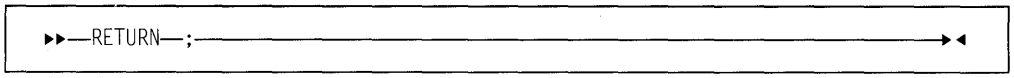
References and expressions in the CALL statement are evaluated in the block in which the call is executed. This includes execution of any ON-units entered as the result of the evaluations.

RETURN statement

The RETURN statement terminates execution of the subroutine or function procedure that contains the RETURN statement and returns control to the invoking procedure. Control is returned to the point immediately following the invocation reference.

Return from a subroutine

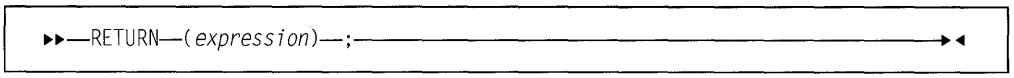
To return from a subroutine, the RETURN statement syntax is:



If the RETURN statement terminates the main procedure, the FINISH condition is raised prior to program termination.

Return from a function

To return from a function, the RETURN statement syntax is:

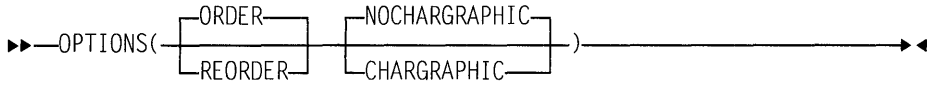


The value returned to the function reference is the value of the expression specified, converted to conform to the attributes specified in the RETURNS option of the procedure statement. You cannot specify an expression for the RETURN statement in a BEGIN block.

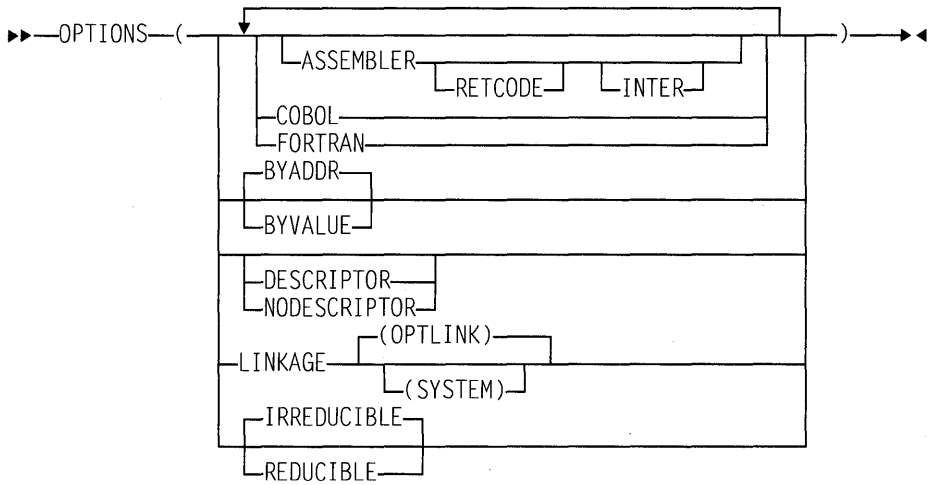
OPTIONS option and attribute

The OPTIONS option can be specified on PACKAGE, PROCEDURE, and BEGIN statements. The OPTIONS attribute can be specified on ENTRY declarations. It is used to specify processing characteristics that apply to the block and the invocation of a procedure. The options shown in the following syntax diagrams are listed alphabetically and discussed starting on page 123.

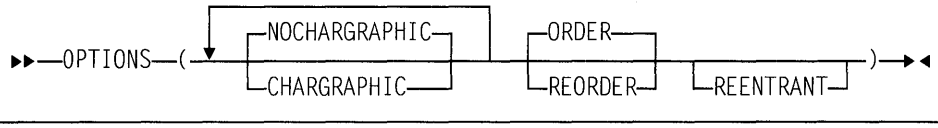
BEGIN statement



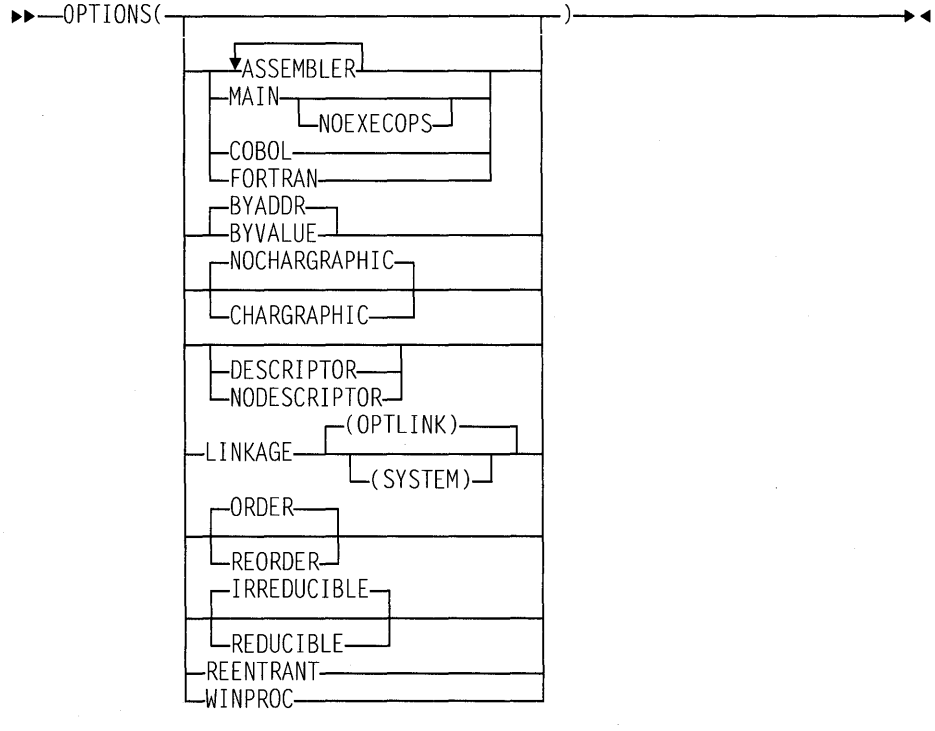
ENTRY declaration



PACKAGE statement



PROCEDURE statement



The options are separated by blanks or commas. They can appear in any order.

ASSEMBLER

Abbreviation: ASM

The ASSEMBLER option is a synonym for NODESCRIPTOR.

BYADDR or BYVALUE

specify how arguments and parameters are passed and received. BYADDR is the default.

BYVALUE may be specified only for scalar arguments and parameters that have known lengths and sizes.

The BYVALUE and BYADDR attributes can also be specified in the description list of an entry declaration and in the attribute list of a parameter declaration. Specifying BYVALUE or BYADDR in an entry or a parameter declaration overrides the option specified in an OPTIONS statement.

The following examples show BYVALUE and BYADDR in both entry declarations and in the OPTIONS statement. The examples assume that the compiler option DEFAULT(BYADDR) is in effect.

Example 1

```
dcl D entry (fixed bin byaddr,
             ptr,
             char(4) byvalue)        /*  byvalue not needed      */
      options(byvalue);

D: proc(I, P, C)
   options(byvalue);
  dcl I fixed bin byaddr,
      P ptr,
      C char(4) byvalue;

  call E2(P);                        /*  P is passed BYADDR      */
```

Example 2

```
dcl F entry (fixed bin byaddr,       /*  byaddr not needed       */
             ptr,
             char(4) byvalue)
      options(byaddr);

F: proc(I,P,C) options(byaddr);
  dcl I fixed bin byaddr;            /*  byaddr not needed       */
  dcl P ptr byaddr;                  /*  byaddr not needed       */
  dcl C char(4) byvalue;             /*  byvalue needed          */

  call E3(C);                        /*  C is passed by address  */

  dcl E4 entry(fixed bin byvalue);

  call E4(I);                        /*  I is passed by value    */
```

CHARGRAPHIC or NOCHARGRAPHIC

Abbreviations: CHARG, NOCHARG

The default for an external procedure is NOCHARG. Internal procedures and begin blocks inherit their defaults from the containing procedure.

When CHARG is in effect, the following semantic changes occur:

- All character string assignments are considered to be mixed character assignments.
- STRINGSIZE condition causes MPSTR built-in function to be used. STRINGSIZE must be enabled for character assignment that can cause truncation and intelligent DBCS truncation is required. (For information on the MPSTR BUILTIN see "MPSTR" on page 415.) For example:

```
Name: procedure options(chargraphic;
      dcl A char(5);
      dcl B char(8);

/* the following statement...                              */

      (stringsize): A=B;

/*...is logically transformed into...                      */

      A=mpstr(B,'vs',length(A));
```

When NOCHARG is in effect, no semantic changes occur.

COBOL
has the same effect as NODESCRIPTOR.

DESCRIPTOR or NODESCRIPTOR
indicates whether the procedure specified in the entry declaration or procedure statement will be passed a descriptor list when it is invoked.

If DESCRIPTOR appears, the compiler will pass a descriptor list, if necessary.

If NODESCRIPTOR appears, the compiler will not pass a descriptor list.

If neither appears, DESCRIPTOR is assumed only when one of the invoked procedure's parameters is a string, array, area, structure, or union.

It is an error for NODESCRIPTOR to appear on a procedure statement or entry declaration in which any of the parameters or elements use the asterisk (*) to indicate the extents, length, or size, or if any parameter is NONCONNECTED.

FORTRAN
has the same effect as NODESCRIPTOR.

LINKAGE
specifies the calling convention to be used for the procedure.

OPTLINK
is the default, and provides a faster alternative to the SYSTEM linkage convention. It is not standard for all OS/2 applications, however.

SYSTEM
is the calling convention normally used for calls to the operating system. Although it is slower than OPTLINK, it is standard for all OS/2 applications and is used for calling OS/2 application programming interfaces.

For more information about calling conventions, refer to *PL/I Package/2 Programming Guide*.

MAIN
indicates that this external procedure is the initial procedure of a PL/I program. MAIN is valid, and required, only on one external procedure per program. The operating-system control program invokes it as the first step in the execution of that program.

NOEXECOPS
The NOEXECOPS option is valid only with the MAIN option. It specifies that the run-time options will not be specified on the command or statement that invokes the program. Only parameters for the main procedure will be specified.

ORDER or REORDER
ORDER and REORDER are optimization options that are specified for a procedure or begin block.

ORDER indicates that only the most recently assigned values of variables modified in the block are available for ON-units that are entered because of computational conditions raised during statement execution and expressions in the block.

The REORDER option allows the compiler to generate optimized code to produce the result specified by the source program when error-free execution takes place.

For more information on using the ORDER and REORDER options, refer to *PL/I Package/2 Programming Guide.*

If neither option is specified for the external procedure, the default is set by the DEFAULT compiler option. Internal blocks inherit ORDER or REORDER from the containing block.

REDUCIBLE or IRREDUCIBLE
Abbreviations: RED, IRRED

REDUCIBLE indicates that a procedure or entry need not be invoked multiple times if the argument(s) stays unchanged, and that the invocation of the procedure has no side effects.

For example, a user-written function that computes a result based on unchanging data should be declared REDUCIBLE. A function that computes a result based on changing data, such as a random number or time of day, should be declared IRREDUCIBLE.

REENTRANT
PL/I compiler for OS/2 programs are always reentrant.

RETCODE
has no effect.

WINPROC
indicates that this procedure is a window procedure in a Presentation Manager* (PM) application. It gets control from PM and returns control to it. If any exceptions occur in this procedure or any of its descendants, normal condition handling actions will be taken as if this were the main procedure. That is, only on-units established in this procedure and any blocks activated by this procedure will be able to get control.

The PL/I implicit action following the FINISH condition normally results in terminating the program, but in this case, the implicit action is to return normally to PM with an appropriate return code.

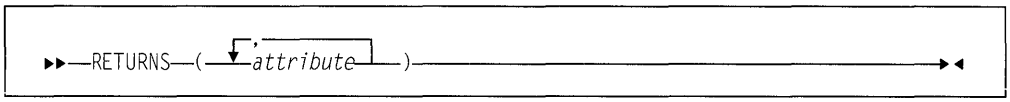
The WINPROC option must be used to avoid system crashes if a window procedure could incur an exception and does have an not an appropriate ON-unit to handle it.

RETURNS option and attribute

If a procedure is a function procedure, you must specify the RETURNS option on the procedure statement. Further, in the invoking procedure or package, you must declare such a procedure as an entry with the RETURNS attribute. The RETURNS option and the RETURNS attribute are used to specify the attributes of the value that is being returned. The attributes in the RETURNS option must match the attributes in the RETURNS attribute.

Procedures that are subroutines (and are therefore invoked using the CALL statement) must not have the RETURNS option on the procedure statement and their entry declaration must not have the RETURNS attribute.

The syntax for RETURNS is:



The attributes are specified in the same way as they are in a declare statement. Defaults are applied in the normal way.

The attributes that can be specified are any of the data attributes and alignment attributes for scalar variables as shown in Figure 10 on page 29. ENTRY variables must have the LIMITED attribute.

String lengths and area sizes must be specified by constants. The returned value has the specified length or size.

Chapter 7. Data declaration

Chapter 7. Data declaration

When a PL/I program is executed, it can manipulate many different data items. Each data item, except a literal arithmetic or string constant, is referred to in the program by a name. Each data name is given attributes and a meaning by a declaration (explicit or implicit).

Most attributes of data items are known at the time the program is compiled. For nonstatic items, attribute values (the bounds of the dimensions of arrays, the lengths of strings, area sizes, initial values) and some file attributes can be determined during execution of the program. Refer to "Block activation" on page 91 for more information.

Data items, types, and attributes are introduced in Chapter 3, "Data elements" on page 24

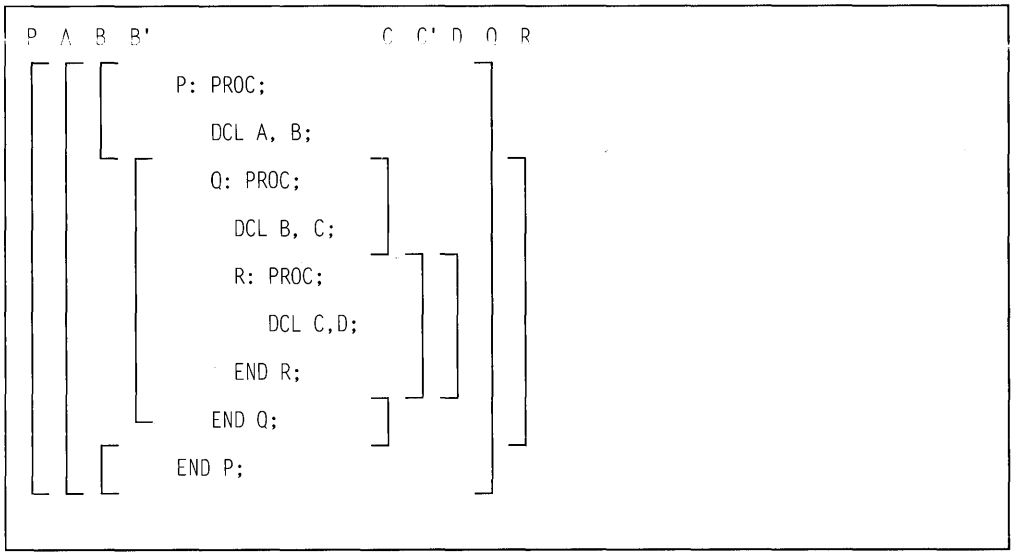
This chapter discusses explicit and implicit declarations, scalar, array, structure, and union declarations, scope of names, data alignment, and default attributes.

Explicit declaration

A name is explicitly declared if it appears:

- In a DECLARE statement. The DECLARE statement explicitly declares attributes of names.
- As an entry constant. Labels of PROCEDURE statements constitute declarations of the entry constants within the containing procedure.
- As a label constant. A label constant explicitly declares a label.
- As a format constant. A label on a FORMAT statement constitutes an explicit declaration of the label.

The scope of an explicit declaration of a name is the block containing the declaration. This includes all contained blocks, except those blocks (and any blocks contained within them) to which another explicit declaration of the same name is internal. In the following diagram, the lines indicate the scope of the declaration of the names.



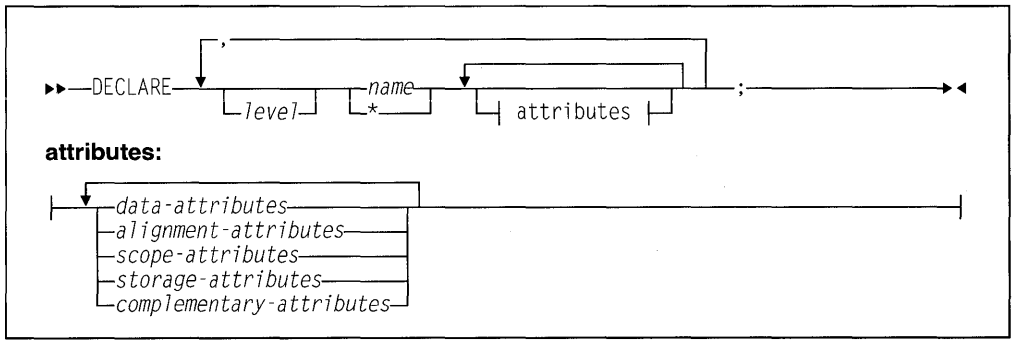
B and B' indicate the two distinct uses of the name B; C and C' indicate the two uses of the name C.

For more information about scope, refer to "Scope of declarations" on page 132.

DECLARE statement

The DECLARE statement specifies some or all of the attributes of a name. If the attributes are not explicitly declared and cannot be determined by context, default attributes are applied.

DECLARE statements can be an important part of the documentation of a program. Consequently, you can make liberal use of declarations, even when default attributes suffice or when an implicit declaration is possible. Because there are no restrictions on the number of DECLARE statements, you can use different DECLARE statements for different groups of names. Any number of names can be declared in one DECLARE statement.



Abbreviation: DCL

For more information about declaring arrays, structures, and unions, refer to "Arrays" on page 144 or "Structures" on page 147 or "Unions" on page 149.

* cannot be used as the *name* of an INTERNAL or an EXTERNAL scalar or as the name of a level 1 EXTERNAL structure or union unless the EXTERNAL attribute specifies an environment name.

attributes

The attributes can appear in any order.

All attributes given explicitly for the name must be declared together in a DECLARE statement, except that:

- Names having the FILE attribute can also be given attributes in an OPEN statement (or have attributes implied by an implicit opening). For more information on the OPEN statement, see "OPEN statement" on page 236.
- The parameter attribute is contextually applied by the appearance of the name in a parameter list. A DECLARE statement internal to the block can specify additional attributes.

Attributes of external names, in separate blocks and compilations, must be consistent.

For more information about attributes and the members of the given groups, refer to "Data types and attributes" on page 25.

level A nonzero integer. If a level number is not specified, *level 1* is the default for element and array variables. *Level 1* must be specified for major structure and union names.

name Each level-1 name must be unique within a block. For more information on level-1 names, refer to "Structures" on page 147.

Condition prefixes and labels cannot be specified on a DECLARE statement.

Factoring attributes

Attributes common to several names can be factored to eliminate repeated specification of the same attributes. Factoring is achieved by enclosing the names in parentheses followed by the set of attributes which apply to all of the names. Factoring can be nested. The dimension attribute can be factored. Factoring can also be used on elementary names within structures and unions. A factored level number must precede the parenthesized list.

Names within the parenthesized list are separated by commas. No factored attribute can be overridden for any of the names, but any name within the list can be given other attributes as long as there is no conflict with the factored attributes.

The following examples show factoring. The last declaration in the set of examples shows nested factoring.

```
declare (A,B,C,D) binary fixed (31);

declare (E decimal(6,5), F character(10)) static;

declare 1 A, 2(B,C,D) (3,2) binary fixed (15);

declare ((A,B) fixed(10),C float(5)) external;
```

Implicit declaration

If a name appears in a program and is not explicitly declared, it is implicitly declared. The scope of an implicit declaration is determined as if the name were declared in a DECLARE statement immediately following the PROCEDURE statement of the external procedure in which the name is used.

With the exception of files, entries, and built-in functions, implicit declaration has the same effect as if the name were declared in the outermost procedure. For files and built-in functions, implicit declaration has the same effect as if the names were declared in the logical package outside any procedures.

Note: Using implicit declarations for anything other than built-in functions and the files SYSIN and SYSPRINT is in violation of the 1987 ANSI standard and should be avoided.

Some attributes for a name declared implicitly can be determined from the context in which the name appears. These cases, called *contextual declarations*, are:

- A name of a built-in function.
- A name that appears in a CALL statement or the CALL option of INITIAL, or that is followed by an argument list, is given the ENTRY and EXTERNAL attributes.
- A name that appears in the parameter list of a PROCEDURE statement is given the PARAMETER attribute.
- A name that appears in a FILE or COPY option, or a name that appears in an ON, SIGNAL, or REVERT statement for a condition that requires a file name, is given the FILE attribute.
- A name that appears in an ON CONDITION, SIGNAL CONDITION, or REVERT CONDITION statement is given the CONDITION attribute.
- A name that appears in the BASED attribute, in a SET option, or on the left-hand side of a locator qualification symbol is given the POINTER attribute.
- A name that appears in an IN option, or in the OFFSET attribute, is given the AREA attribute.

Examples of contextual declaration are:

```
read file (PREQ) into (Q);

allocate X in (S);
```

In these statements, `PREQ` is given the FILE attribute, and `S` is given the AREA attribute.

Implicit declarations that are not contextual declarations acquire all attributes by default, as described in "Defaults for attributes" on page 141. Because a contextual declaration cannot exist within the scope of an explicit declaration, it is impossible for the context of a name to add to the attributes established for that name in an explicit declaration.

Scope of declarations

The part of the program to which a name applies is called the *scope of the declaration* of that name. In most cases, the scope of the declaration of a name is determined entirely by the position where the name is declared within the program. Implicit declarations are treated as if the name were declared in a DECLARE statement immediately following the PROCEDURE statement of the external procedure.

It is not necessary for a name to have the same meaning throughout a program. A name explicitly declared within a block has a meaning only within that block. Outside the block, the name is unknown unless the same name has also been declared in the outer block. Each declaration of the name establishes a scope and in this case, the name in the outer block refers to a different data item. This enables you to specify local definitions and, hence, to write procedures or begin-blocks without knowing all the names used in other parts of the program.

In the following example, the output for `A` is actually `C.A`, which is `2`. The output for `B` is `1`, as declared in procedure `X`.

```
X: proc options(main);
 dcl (A,B) char(1) init('1');
  call Y;
  return;

  Y: proc;
    dcl 1 C,
        3 A char(1) init('2');
    put data(A,B);
    return;
  end Y;
end X;
```

Thus, for nested procedures, PL/I uses the variable declared within the current block before using any variables that are declared in containing blocks.

In order to understand the scope of the declaration of a name, you must understand the terms *contained in* and *internal to*.

All of the text of a block, from the PACKAGE, PROCEDURE, or BEGIN statement through the corresponding END statement (including condition prefixes of BEGIN, PACKAGE, and PROCEDURE statements), is said to be contained in that block. However, the labels of the BEGIN or PROCEDURE statement heading the block are not contained in that block. Nested blocks are contained in the block in which they appear.

Text that is contained in a block, but not contained in any other block nested within it, is said to be internal to that block. Entry names of a procedure (and labels of a BEGIN statement) are not contained in that block. Consequently, they are internal to the containing block. Entry names of an external procedure are treated as if they were external to the external procedure.

Figure 28 illustrates the scopes of data declarations.

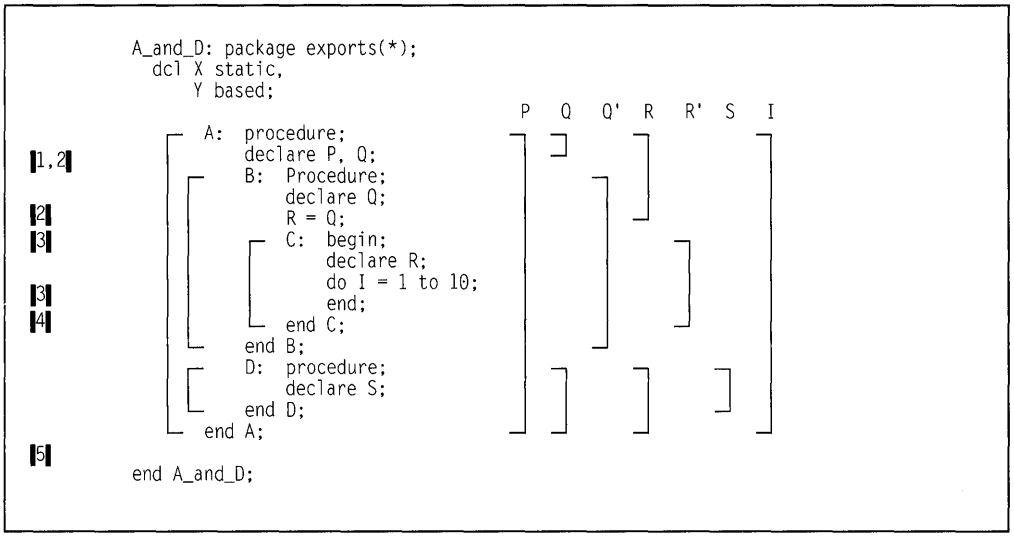


Figure 28. Scopes of data declarations

The brackets to the left indicate the block structure; the brackets to the right show the scope of each declaration of a name. The scopes of the two declarations of Q and R are shown as Q and Q' and R and R'.

Note that X and Y are visible to all of the procedures contained in the package.

1 P is declared in the block A and known throughout A because it is not redeclared.

2 Q is declared in block A, and redeclared in block B. The scope of the first declaration of Q is all of A except B; the scope of the second declaration of Q is block B only.

3 R is declared in block C, but a reference to R is also made in block B. The reference to R in block B results in an implicit declaration of R in A, the external procedure. Therefore, two separate names (R and R' in Figure 28) with different scopes exist. The scope of the explicitly declared R is block C; the scope of the implicitly declared R in block B is all of A except block C.

4 I is referred to in block C. This results in an implicit declaration in the external procedure A. As a result, this declaration applies to all of A, including the contained procedures B, C, and D.

5 S is explicitly declared in procedure D and is known only within D.

Figure 29 illustrates the scopes of entry constant and statement label declarations.

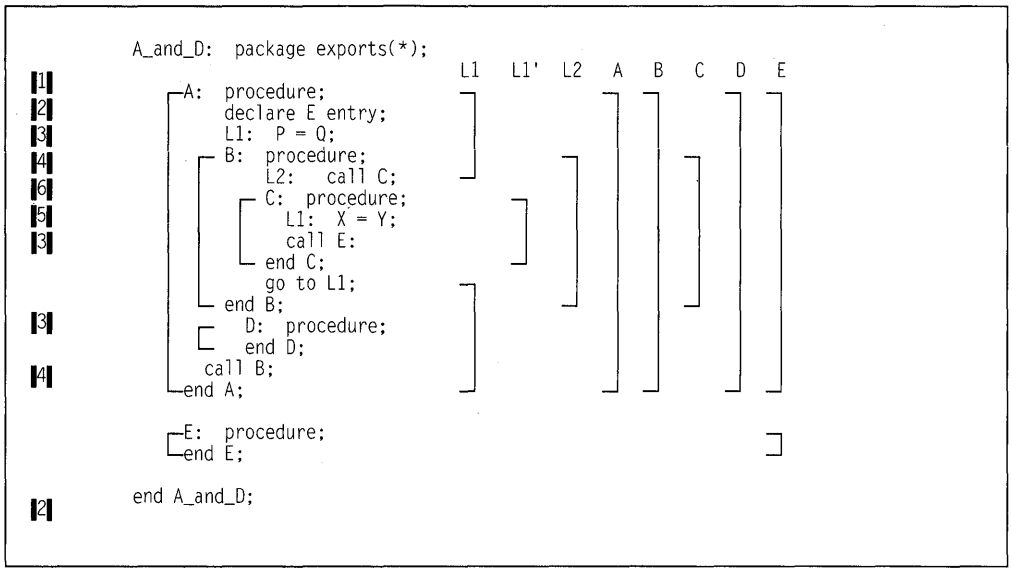


Figure 29. Scopes of entry and label declarations

Figure 29 shows two external procedures, A and E.

|1| The scope of the declaration of the name A is only all of the block A, and not E.

|2| E is explicitly declared in A as an external entry constant. The explicit declaration of E applies throughout block A. It is not linked to the explicit declaration of E that applies throughout block E. The scope of the declaration of the name E is all of block A and all of block E.

|3| The label L1 appears with statements internal to A and to C. Two separate declarations are therefore established; the first applies to all of block A except block C, the second applies to block C only. Therefore, when the GO TO statement in block B executes, control transfers to L1 in block A, and block B terminates.

|4| D and B are explicitly declared in block A and can be referred to anywhere within A; but because they are INTERNAL, they cannot be referred to in block E.

|5| C is explicitly declared in B and can be referred to from within B, but not from outside B.

|6| L2 is declared in B and can be referred to in block B, including C, which is contained in B, but not from outside B.

INTERNAL and EXTERNAL attributes

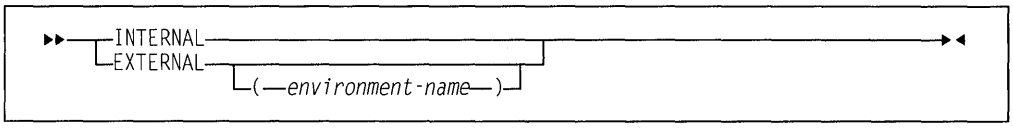
The INTERNAL and EXTERNAL attributes define the scope of a name.

INTERNAL specifies that the name can be known only in the declaring block. Any other explicit declaration of that name refers to a new object with a different scope that does not overlap.

A name with the EXTERNAL attribute can be declared more than once, either in different external procedures or within blocks contained in external procedures. All declarations of the same name with the EXTERNAL attribute refer to the same data. The scope of each declaration of the name (with the EXTERNAL attribute)

includes the scopes of all the declarations of that name (with EXTERNAL) within the application.

The syntax for the INTERNAL and EXTERNAL attributes is:



Abbreviations: INT for INTERNAL, EXT for EXTERNAL

environment-name
specifies the name by which the procedure or variable is known outside of the compilation unit.

When so specified, the name being declared effectively becomes internal and is not known outside of the compilation unit. The environment name is known instead.

Note: Use of environment names can limit the portability of your application.

INTERNAL is the default for entry names of internal procedures and for variables with any storage class except controlled.

EXTERNAL is the default for file constants, entry constants, programmer-defined conditions, and controlled variables.

When a major structure or union name is declared EXTERNAL in more than one block, the attributes of the members must be the same in each case, although the corresponding member names need not be identical.

In the following example:

```
ProcA: procedure;
  declare 1 A external,
            2 B,
            2 C;
            ⋮

end ProcA;

%process;
ProcB: procedure;
  declare 1 A external,
            2 B,
            2 D;
            ⋮
end ProcB;
```

If `A.B` is changed in `ProcA`, it is also changed for `ProcB`, and vice versa; if `A.C` is changed in `ProcA`, `A.D` is changed for `ProcB`, and vice versa.

Members of structures and unions always have the INTERNAL attribute.

Because external declarations for the same name all refer to the same data, they must all result in the same set of attributes. When EXTERNAL names are declared in different external procedures, the user has the responsibility to ensure that the attributes are matching. Figure 30 illustrates a variety of declarations and their scopes.

```
        Scope_Example:  package exports(*);
|1|       A: procedure;
|2|          declare S character (20);
|7|          dcl Set entry(fixed decimal(1)),
|7|              Out entry(label);
             call Set (3);
|9|         E: get list (S,M,N);
|8|         B: begin;
|4,5|          declare X(M,N), Y(M);
              get list (X,Y);
              call C(X,Y);

|9,5|        C:  procedure (P,Q);
                 declare
                   P(*,*),
                   Q(*),
|12,2|              S binary fixed external;
                 S = 0;
|6|              do I = 1 to M;
                   if sum (P(I,*)) = Q(I) then
|8|                   go to B;
                   S = S+1;
                   if S = 3 then
|9|                   call Out (E);
                   Call D(I);
|8|         B:  end;
            end C;

|9|         D:  procedure (N);
               put list ('Error in row ',
|2,3|                 N, 'Table Name ', S);
            end D;
            end B;
            go to E;
            end A;

|9|        Out:  procedure (R);
              Declare
                R Label,
|11|            (K static internal,
|11,7|             L static external) init (0),
|12|            S binary fixed external,
                Z fixed decimal(1);
              K = K+1; S=0;
              if K<L then
                stop;
|10|           else go to R;
           end;
           Set:  procedure (Z);
              declare Z fixed dec(1);
|7|           L=Z;
              declare L external init(0);
              return;
              end;
         end Scope_Example;
```

Figure 30. Example of scopes of various declarations

|1| A is an external procedure name. Its scope is all of block A, plus any other blocks where A is declared as external.

|2| S is explicitly declared in block A and block C. The character variable declaration applies to all of block A except block C. The fixed binary declaration applies only within block C. Notice that although D is called from within block C, the reference to S in the PUT statement in D is to the character variable S, and not to the S declared in block C.

|3| N appears as a parameter in block D, but is also used outside the block. Its appearance as a parameter establishes an explicit declaration of N within D. The references outside D cause an implicit declaration of N in block A. These two declarations of the name N refer to different objects, although in this case, the objects have the same data attributes, which are, by default, FIXED BIN(15,0) and INTERNAL. Under DFT(ANS), the precision is (31,0).

|4| X and Y are known throughout B and can be referred to in block C or D within B, but not in that part of A outside B.

|5| P and Q are parameters, and therefore if there were no other declaration of these names within the block, their appearance in the parameter list would be sufficient to constitute a contextual declaration. However, a separate, explicit declaration statement is required in order to specify that P and Q are arrays. Although the arguments X and Y are declared as arrays and are known in block C, it is still necessary to declare P and Q in a DECLARE statement to establish that they, too, are arrays. (The asterisk notation indicates that the bounds of the parameters are the same as the bounds of the arguments.)

|6| I and M are not explicitly declared in the external procedure A. Therefore, they are implicitly declared and are known throughout A, even though I appears only within block C.

|7| The Out and Set external procedures in the example have an external declaration of L that is common to both. They also must be declared explicitly with the ENTRY attribute in procedure A. Because ENTRY implies EXTERNAL, the two entry constants Set and Out are known throughout the two external procedures.

|8| The label B appears twice in the program—first in A, as the label of a begin-block, which is an explicit declaration, and then redeclared as a label within block C by its appearance as a prefix to an END. statement. The go to B statement within block C, therefore, refers to the label of the END statement within block C. Outside block C, any reference to B is to the label of the begin block.

|9| Blocks C and D can be called from any point within B but not from that part of A outside B, nor from another external procedure. Similarly, because label E is known throughout the external procedure A, a transfer to E can be made from any point within A. The label B within block C, however, can be referred to only from within C. Transfers out of a block by a GO TO statement can be made; but such transfers into a nested block generally cannot. An exception is shown in the external procedure Out, where the label E from block C is passed as an argument to the label parameter R.

Note that, with no files specified in the GET and PUT statements, SYSIN and SYSPRINT are implicitly declared.

|10| The statement else go to R; transfers control to the label E, even though E is declared within A, and not known within Out.

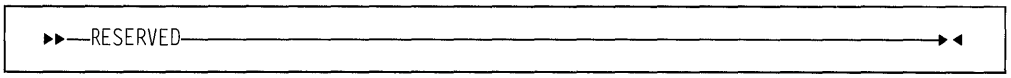
|11| The variables K (INTERNAL) and L (EXTERNAL) are declared as STATIC within the Out procedure block; their values are preserved between calls to Out.

|12| In order to identify the S in the procedure Out as the same S in the procedure C, both are declared with the attribute EXTERNAL.

RESERVED attribute

RESERVED indicates that the compilation unit containing the declared variable is linked with at least one other compilation unit that declares the same variable with the STATIC and EXTERNAL attributes and without the RESERVED attribute. The RESERVED attribute implies STATIC and EXTERNAL attributes, but does not allocate or initialize storage for the variable.

The syntax for the RESERVED attribute is:



Data alignment

The computer holds information in multiples of units of 8 bits. Each 8-bit unit of information is called a *byte*.

The computer accesses bytes singly or as halfwords, words, or doublewords. A *halfword* is 2 consecutive bytes. A *fullword* is 4 consecutive bytes. A *doubleword* is 8 consecutive bytes. Byte locations in storage are consecutively numbered starting with 0; each number is the address of the corresponding byte. Halfwords, words, and doublewords are addressed by the address of their leftmost byte.

Your programs can execute faster if halfwords, words, and doublewords are located in main storage on an integral boundary for that unit of information. That is, the unit of information's address is a multiple of the number of bytes in the unit, as can be seen in Figure 31.

Figure 31. Alignment on integral boundaries of halfwords, words, and doublewords

ADDRESSES IN A SECTION OF STORAGE							
5000	5001	5002	5003	5004	5005	5006	5007
byte	byte	byte	byte	byte	byte	byte	byte
halfword		halfword		halfword		halfword	
fullword				fullword			
doubleword							

PL/I permits data alignment on integral boundaries. However, unused bytes between successive data elements can increase storage use. For example, when the data items are members of aggregates used to create a data set, the unused bytes increase the amount of auxiliary storage required. The ALIGNED and UNALIGNED attributes allow you to choose whether or not to align data on the appropriate integral boundary.

ALIGNED and UNALIGNED attributes

ALIGNED specifies that the data element is aligned on the storage boundary corresponding to its data-type requirement. These requirements are shown in Figure 32.

Figure 32 (Page 1 of 2). Alignment requirements

Variable Type	Stored Internally as:	Storage Requirements (Bytes)	Alignment Requirements	
			ALIGNED Data	UNALIGNED Data
BIT (n)	ALIGNED: One byte for each group of 8 bits (or part thereof) UNALIGNED: As many bits as are required, regardless of byte boundaries	ALIGNED: CEIL(n/8) UNALIGNED: n bits	Byte (Data may begin on any byte, 0 through 7)	Bit (Data may begin on any bit in any byte, 0 through 7)
CHARACTER (n)	One byte per character	n		Byte (Data may begin on any byte, 0 through 7)
GRAPHIC (n)	Two bytes per graphic	2n		
PICTURE	One byte for each PICTURE character (except V, K, and the F scaling factor specification)	Number of PICTURE characters other than V, K, and F specification		
DECIMAL FIXED (p,q)	Packed decimal format (1/2 byte per digit, plus 1/2 byte for sign)	CEIL((p+1)/2		
BINARY FIXED(p,q) SIGNED 1 <= p <= 7 UNSIGNED 1 <= p <= 8	One byte	1		
BIT(n) VARYING	Two-byte prefix plus 1 byte for each group of 8 bits (or part thereof) of the declared maximum length	ALIGNED: 2+CEIL(n/8) UNALIGNED: 2 bytes + n bits	Halfword (Data may begin on byte 0, 2, 4, or 6)	
CHARACTER(n) VARYING	Two-byte prefix plus 1 byte per character of the declared maximum length	2+n		
GRAPHIC(n) VARYING	Two-byte prefix plus 2 bytes per graphic of the declared maximum length	2+2n		
BINARY FIXED(p,q) SIGNED 7 <= p <= 15 UNSIGNED 8 <= p <= 16	Halfword	2		
BINARY FIXED(p,q) SIGNED 15 <= p <= 31 UNSIGNED 16 <= p <= 31	Fullword	4	Fullword (Data may begin on byte 0 or 4)	
BINARY FLOAT (p) 1<=p<=21	Short floating-point			
DECIMAL FLOAT (p) 1<=p<=6				
POINTER	–	4	Fullword (Data may begin on byte 0 or 4)	Byte (Data may begin on any byte, 0 through 7)
OFFSET	–			
FILE	–			
ENTRY LIMITED	–			
ENTRY	–	8		
LABEL or FORMAT	–			

Alignment and storage requirements for program control data can vary across supported systems.

Complex data requires twice as much storage as its real counterpart, but the alignment requirements are the same.

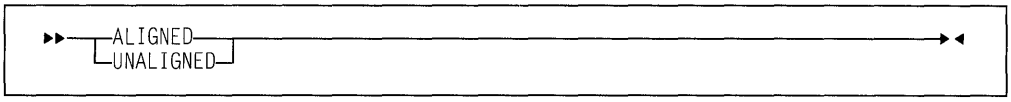
Figure 32 (Page 2 of 2). Alignment requirements

Variable Type	Stored Internally as:	Storage Requirements (Bytes)	Alignment Requirements	
			ALIGNED Data	UNALIGNED Data
AREA	–	16+size	Doubleword (Data may begin on byte 0)	AREA data cannot be unaligned
BINARY FLOAT (p) 22<=p<=53	Long floating-point	8		Byte (Data may begin on any byte, 0 through 7)
DECIMAL FLOAT (p) 7<=p<=16				
BINARY FLOAT(p) 54<=p<=64	Extended floating-point			
DECIMAL FLOAT(p) 17<=p<=18				

Alignment and storage requirements for program control data can vary across supported systems.

Complex data requires twice as much storage as its real counterpart, but the alignment requirements are the same.

UNALIGNED specifies that each data element is mapped on the next byte boundary, except for fixed-length bit strings, which are mapped on the next bit. The syntax for the ALIGNED and UNALIGNED attributes is:



Defaults are applied at element level. UNALIGNED is the default for bit data, character data, graphic data, and numeric character data. ALIGNED is the default for all other types of data.

ALIGNED or UNALIGNED can be specified for element, array, structure or union variables. The application of either attribute to a structure or union is equivalent to applying the attribute to all contained elements that are not explicitly declared ALIGNED or UNALIGNED.

The following example illustrates the effect of ALIGNED and UNALIGNED declarations for a structure and its elements:

```
declare 1 S,
          2 X bit(2),       /* unaligned by default  */
          2 A aligned,      /* aligned explicitly    */
            3 B,            /* aligned from A        */
            3 C unaligned,  /* unaligned explicitly  */
              4 D,          /* unaligned from C      */
              4 E aligned,  /* aligned explicitly    */
              4 F,          /* unaligned from C      */
            3 G,            /* aligned from A        */
          2 H;              /* aligned by default    */
```

For more information about structures and unions, refer to "Structures" on page 147 and "Unions" on page 149.

Defaults for attributes

Every name in a PL/I program requires a complete set of attributes. Arguments passed to a procedure must have attributes matching the procedure's parameters. Values returned by functions must have the attributes expected. However, the attributes that you specify need rarely include the complete set of attributes.

The set of attributes for:

- Explicitly declared names
- Implicitly (including contextually) declared names
- Attributes to be included in parameter descriptors
- Values returned from function procedures

can be completed by using the language-specified defaults, or by defaults that you can define (using the DEFAULT statement) either to modify the language-specified defaults or to develop a completely new set of defaults.

Attributes applied by default cannot override attributes applied to a name by explicit or contextual declaration.

Language-specified defaults

When a variable has not been declared with any data attributes, it is given arithmetic attributes by default. If mode, scale and base are not specified by a DECLARE or DEFAULT statement, the DEFAULT compiler option determines its attributes as follows:

- If DEFAULT(IBM) is in effect, variables with names beginning with the letters I through N are given the attributes REAL FIXED BIN(15,0); all other variables are given the attributes REAL FLOAT DEC(6).
- If DEFAULT(ANS) is in effect, all variables are given the attributes REAL FIXED BIN(31,0)

If a scaling factor is specified in the precision attribute, the attribute FIXED is applied before any other attributes. Therefore, a declaration with the attributes BIN(p,q) is always equivalent to a declaration with the attributes FIXED BIN(p,q).

If a precision is not specified in an arithmetic declaration, the DEFAULT compiler option determines the precision as indicated in Figure 33 on page 142. The language-specified defaults for scope, storage and alignment attributes are shown in Figure 10 on page 29 and Figure 9 on page 28.

If no description list is given in an ENTRY declaration, the attributes for the argument must match those specified for the corresponding parameter in the invoked procedure. For example, given the following declaration:

```
dcl x entry;
call x( 1 );
```

The argument has the attributes REAL FIXED DEC(1,0). This would be an error if the procedure x declared its parameter with other attributes, as shown in the following example:

```
x: proc( y );
   dcl y fixed bin(15);
```

This potential problem can be easily avoided if the entry declaration specifies the attributes for all of its parameters.

Figure 33. Default arithmetic precisions

Attributes	DEFAULT(IBM)	DEFAULT(ANS)
DECIMAL FIXED	(5,0)	(10,0)
BINARY FIXED	(15,0)	(31,0)
DECIMAL FLOAT	(6)	(6)
BINARY FLOAT	(21)	(21)

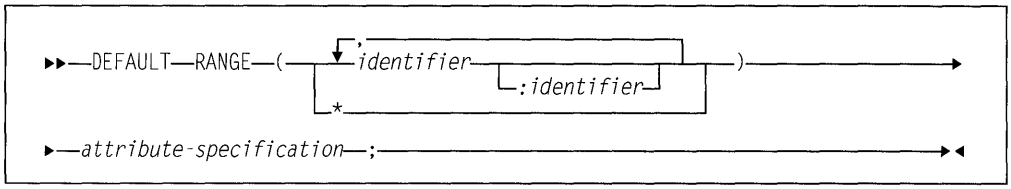
DEFAULT statement

The DEFAULT statement specifies data-attribute defaults (when attribute sets are not complete). Any attributes not applied by the DEFAULT statement for any partially-complete explicit or contextual declarations, and for implicit declarations, are supplied by language-specified defaults.

The DEFAULT statement overrides all other attribute specifications.

Structure and union elements are given default attributes according to the name of the element, not the qualified element name. The DEFAULT statement cannot be used to create a structure or a union.

The syntax for the DEFAULT statement is:



Abbreviation: DFT

RANGE(identifier)

specifies that the defaults apply to names that begin with or exactly match the identifier specified. Identifier can be a complete or partial identifier. For example:

```
range (ABC)
```

applies to these names:

```
ABC
abcd
ABCDE
```

but not to:

```
ABD
acb
AB
A
```

Hence, a single character in the range-specification applies to all names that start with that character.

RANGE(identifier:identifier)

specifies that the defaults apply to names that begin with or exactly match the identifiers in the range specified. Identifier can be a complete or partial identifier but should not have any DBCS characters. The second identifier in the range must be greater than or equal to the first identifier. For example:

```
range(A:C, Fred : Harry)
```

applies to these names:

```
ABC
BCA
C
FRED
George
Harrison
```

but not to:

```
DBA
FLAG
HarrysWorld
```

RANGE(*)

specifies all names (including * names) in the scope of the DEFAULT statement. For example:

```
dft range (*) fixed bin;
```

This statement specifies default attributes FIXED BIN, with default precision for all names.

attribute-specification

specifies a list of attributes from which selected attributes are applied to names in the specified range. (Attributes are listed in "Data types and attributes" on page 25.) Attributes in the list can appear in any order and must be separated by blanks.

Only those attributes that are necessary to complete the declaration of a data item are taken from the list of attributes.

Attributes that conflict, when applied to a data item, do not necessarily conflict when they appear in an attribute specification. Consider the following statement:

```
default range(S) binary varying;
```

Given the declaration `dcl s1 real`, `s1` would get the BINARY attribute, but with the declaration `dcl s2 bit`, `s2` would get the VARYING attribute.

There can be more than one DEFAULT statement within a block. The scope of a DEFAULT statement is the block in which it occurs, and all blocks within that block which neither include another DEFAULT statement with the same range, nor are contained in a block having a DEFAULT statement with the same range.

A DEFAULT statement in an internal block affects only explicitly declared names. This is because the scope of an implicit declaration is determined as if the names were declared in a DECLARE statement immediately following the PROCEDURE statement of the external procedure in which the name appears.

It is possible for a containing block to have a DEFAULT statement with a range that is partly covered by the range of a DEFAULT statement in a contained block. In such a case, the range of the DEFAULT statement in the containing block is reduced by the range of the DEFAULT statement in the contained block. For example:

```
P:  procedure;
    default range (XY) fixed;      /* First  */
Q:  begin;
    default range (XYZ) float;     /* Second */
    end P;
```

The scope of the first DEFAULT statement is procedure P and the contained block Q. The range of the first DEFAULT statement is all names in procedure P beginning with the characters XY, together with all names in begin block Q beginning with the characters XY, except for those beginning with the characters XYZ.

Restoring language-specified defaults

The following statement:

```
dft range(*) system;
```

overrides, for all names, any programmer-defined default rules established in a containing block. It can be used to restore language-specified defaults for contained blocks.

Arrays

An *array* is an n-dimensional collection of elements that have identical attributes. Only the array itself is given a name. An individual item of an array is referred to by giving its position within the array. You indicate that a name is an *array variable* by providing the dimension attribute.

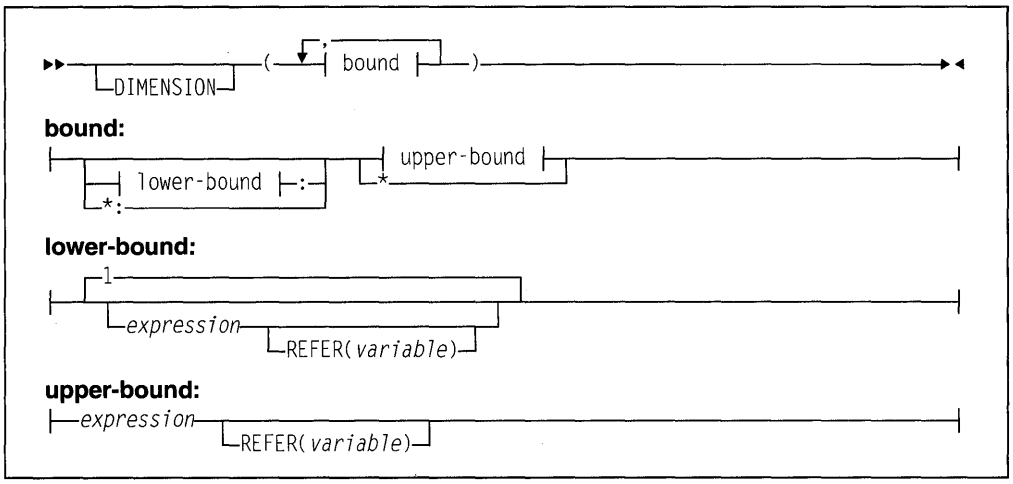
DIMENSION attribute

The *dimension attribute* specifies the number of dimensions of an array and upper and lower bounds of each.

Bounds that are nonrestricted expressions are evaluated and converted to FIXED BINARY(M,0) when storage is allocated for the array.

The *extent* is the number of integers between, and including, the lower and upper bounds.

The syntax for the dimension attribute is:



Abbreviation: DIM

If the DIMENSION keyword is omitted, the dimension must immediately follow the name (or the parenthesized list of names) in the declaration.

The number of bounds specifications indicates the number of dimensions in the array, unless the declared variable is in an array of structures or unions. In this case it inherits dimensions from the containing structure or union.

The bounds specification indicates the bounds as follows:

- If only the upper bound is given, the lower bound defaults to 1.
- The lower bound must be less than or equal to the upper bound.
- An asterisk (*) specifies that the lower and/or the upper bound is to be taken from the argument associated with the parameter.

Examples of arrays

Consider the following declaration:

```
declare List fixed decimal(3) dimension(8);
```

`List` is declared as a one-dimensional array of eight elements, each one a fixed-point decimal element of three digits. The one dimension of `List` has bounds of 1 and 8, and its extent is 8.

In the example:

```
declare Table (4,2) fixed dec (3);
```

`Table` is declared as a two-dimensional array of eight fixed-point decimal elements. The two dimensions of `Table` have bounds of 1 and 4 and 1 and 2, and the extents are 4 and 2.

Other examples are:

```
declare List_A dimension(4:11);
declare List_B (-4:3);
```

In the first example, the bounds are 4 and 11; in the second they are -4 and 3. The extents are the same for each, 8 integers from the lower bound through the upper bound.

In the manipulation of array data (discussed in "Array expressions" on page 66) involving more than one array, the bounds—not merely the extents—must be identical. Although `List`, `List_A`, and `List_B` all have the same extent, the bounds are not identical.

Subscripts

The bounds of an array determine the way elements of the array can be referred to. For example, when the following data items:

```
20 5 10 30 630 150 310 70
```

are assigned to the array `List`, as declared above, the different elements are referred to as follows:

Reference	Element
LIST (1)	20
LIST (2)	5
LIST (3)	10
LIST (4)	30
LIST (5)	630
LIST (6)	150
LIST (7)	310
LIST (8)	70

Each of the parenthesized numbers following LIST is a *subscript*. A parenthesized subscript following an array name reference identifies a particular data item within the array. A reference to a subscripted name, such as LIST(4), refers to a single element and is an element variable. The entire array can be referred to by the unsubscripted name of the array—for example, LIST.

The same data can be assigned to `List_A` and `List_B` declared previously. In this case it is referenced as follows:

Reference	Element	Reference
LIST_A (4)	20	LIST_B (-4)
LIST_A (5)	5	LIST_B (-3)
LIST_A (6)	10	LIST_B (-2)
LIST_A (7)	30	LIST_B (-1)
LIST_A (8)	630	LIST_B (0)
LIST_A (9)	150	LIST_B (1)
LIST_A (10)	310	LIST_B (2)
LIST_A (11)	70	LIST_B (3)

Assume that the same data is assigned to TABLE, which is declared as a two-dimensional array. TABLE can be illustrated as a matrix of four rows and two columns:

TABLE(m,n)	(m,1)	(m,2)
(1,n)	20	5
(2,n)	10	30
(3,n)	630	150
(4,n)	310	70

An element of TABLE is referred to by a subscripted name with two parenthesized subscripts, separated by a comma. For example, TABLE (2,1) would specify the first item in the second row, the data item 10.

The use of a matrix to illustrate TABLE is purely conceptual. It has no relationship to the way the items are actually organized in storage. Data items are assigned to an array in row major order. This means that the subscript that represents rows varies most rapidly. For example, assignment to TABLE would be to TABLE(1,1), TABLE(1,2), TABLE(2,1), TABLE(2,2), and so forth.

A subscripted reference to an array must contain as many subscripts as there are dimensions in the array.

Any expression that yields a valid arithmetic value can be used for a subscript. If necessary, the value is converted to FIXED BINARY(31,0). Thus, TABLE(I,J*K) can be used to refer to the different elements of TABLE by varying the values of I, J, and K.

Cross sections of arrays

Cross sections of arrays can be referred to by using an asterisk for a subscript. The asterisk specifies that the entire extent is used. For example, TABLE(*,1) refers to all of the elements in the first column of TABLE. It specifies the cross section consisting of TABLE(1,1), TABLE(2,1), TABLE(3,1), and TABLE(4,1). The subscripted name TABLE(2,*) refers to all of the data items in the second row of TABLE. TABLE(*,*) refers to the entire array, as does TABLE.

A subscripted name containing asterisk subscripts represents not a single data element, but an array with as many dimensions as there are asterisks. Consequently, such a name is not an element expression, but an array expression.

A reference to a cross section of an array can refer to two or more elements that are not adjacent in storage. The storage represented by such a cross section is known as *nonconnected* storage. (See "CONNECTED and NONCONNECTED attributes" on page 217.) The rule is as follows: if a nonasterisk bound appears to the right of the leftmost asterisk bound, the array cross section is in nonconnected storage. Thus A(4,*,*) is in connected storage; A(*,2,*) is not.

Structures

A structure is a collection of member elements that may be structures, unions, elementary variables and arrays.

The *structure variable* is a name that can be used to refer to the entire aggregate of data. Unlike an array, however, each member of a structure also has a name, and the attributes of each member may differ. An asterisk may be used as the name of a structure or a member when it will not be referred to. For example, reserved or filler items may be named asterisk.

A structure has different *levels*. The name at level-1 is called a *major structure*. Names at deeper levels may be *minor structures or unions*. Names at the deepest level are called *elementary* names, which can represent an elementary variable or an array variable. Unions are described in "Unions" on page 149.

A structure is described in a DECLARE statement through the use of level numbers preceding the associated names. Level numbers must be integers.

A major structure name is declared with the level number 1. Minor structures, unions, and elementary names are declared with level numbers greater than 1. A delimiter (usually a blank) must separate the level number and its associated name. For example, the items of a payroll record could be declared as follows:

```
declare 1 Payroll,                    /*  major structure name */
          2 Name,                     /*  minor structure name */
            3 Last char(20),          /*  elementary name      */
            3 First char(15),
          2 Hours,
            3 Regular fixed dec(5,2),
            3 Overtime fixed dec(5,2),
          2 Rate,
            3 Regular fixed dec(3,2),
            3 Overtime fixed dec(3,2);
```

In the example, `Payroll` is the major structure and all other names are members of this structure. `Name`, `Hours`, and `Rate` are minor structures, and all other members are elementary variables. You can refer to the entire structure by the name `Payroll`, or to portions of the structure by the minor structure names. You can refer to a member by referring to the member name.

Indentation is only for readability. The statement could be written in a continuous string as:

```
Declare 1 Payroll, 2 Name, 3 Last char(20), . . .
```

The level numbers you choose for successively deeper levels need not be consecutive. A minor structure at level `n` contains all the names with level numbers greater than `n` that lie between that minor structure name and the next name with a level number less than or equal to `n`.

The description of a major structure is usually terminated by a semicolon terminating the DECLARE statement. It can also be terminated by comma, followed by the declaration of another item.

For example, the following declaration results in exactly the same structure as the declaration in the previous example.

```
Declare 1 Payroll,
          4 Name,
            5 Last char(20),
            5 First char(15),
          3 Hours,
            6 Regular fixed dec(5,2),
            5 Overtime fixed dec(5,2),
          2 Rate,
            9 Regular fixed dec(3,2),
            9 Overtime fixed dec(3,2);
```

Unions

A union is a collection of member elements that overlay each other, occupying the same storage. The members may be structures, unions, elementary variables, and arrays. They need not have identical attributes.

The entire union is given a name that can be used to refer to the entire aggregate of data. Like a structure, each element of a union also has a name. An asterisk may be used as the name of a union or a member, when it will not be referred to. For example, reserved or filler items may be named asterisk.

Like a structure, a union may be at any level including level 1. All elements of a union at the next deeper level are members of the union and occupy the same storage. The storage occupied by the union is equal to the storage required by the largest member. Normally, only one member is presumed to be active and valid at any time. The determination of which member is active is entirely under programmer control.

A union, like a structure, is declared through the use of level numbers preceding the associated names.

Unions may be used to declare variant records that would typically contain a common part, a selector part, and variant parts. For example, records in an client file may be declared as follows:

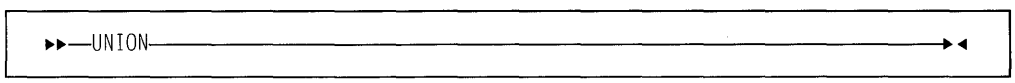
```
Declare 1 Client,
          2 Number pic '999999',
          2 Type bit(1),              /* Client is an individual */
          2 * bit(7),                 /* reserved                */
          2 Name union,
            3 Individual,
              5 Last_Name union,
                7 Last    char(20),
                7 Initial char(1),
              5 First_Name char(15),
            3 Company char(35),
          2 * char(0);
```

In this example, `Client` is a major structure. The structure `Individual`, and the element `Company` are members of the union `Name`. One of these members is active depending on `Type`. The structure `Individual` contains the union `First_name` and the element `Last_name`. `First_name` union has `First` and `Initial` as its members, both of which are active. The example also shows the use of asterisk as a name. The description of a union is terminated by the semicolon that terminates a DECLARE statement or by a comma, followed by the declaration of another item.

UNION attribute

The UNION attribute allows you to specify that a variable is a union and that its members are those that follow it and are at the next logically higher level.

The syntax is:



Structure/union qualification

A member of a structure or a union can be referred to its name alone if it is unique. If another member has the same name, whether at the same or different level, ambiguity occurs. Where ambiguity occurs, a qualified reference is required to uniquely identify the correct member.

A *qualified reference* is a member name that is qualified with one or more names of parent members connected by periods. (See the qualified reference syntax in Chapter 4, "Expressions and references" on page 50.) Blanks may appear surrounding the period.

The qualification must follow the order of levels. That is, the name at the highest level must appear first, with the name at the deepest level appearing last.

While the level 1 structure or union name must be unique within the block scope, member names need not be unique as long as they do not appear at same logical level within their most immediate parent. Qualifying names must be used only so far as necessary to make the reference unique within the block in which it appears. In the following example, the value of `x.y (19)` is displayed, not the value `(17)`.

```
dcl 7 fixed init(17);

begin;
  dcl
   1 x,
     2 y fixed init(19);
  display( y );
end;
```

A reference is always taken to apply to the declared name in the innermost block containing the reference.

The following examples illustrate both ambiguous and unambiguous references. In the following example, `A.C` refers to `C` in the inner block; `D.E` refers to `E` in the outer block.

```
declare 1 A, 2 C, 2 D, 3 E;
   begin;
   declare 1 A, 2 B, 3 C, 3 E;
A.C = D.E;
```

In the following example, `D` has been declared twice. A reference to `A.D` refers to the second `D`, because `A.D` is a complete qualification of only the second `D`. The first `D` is referred to as `A.C.D`.

```
declare 1 A,
          2 B,
          2 C,
            3 D,
          2 D;
```

In the following example, a reference to `A.C` is ambiguous because neither `C` can be completely qualified by this reference.

```
declare 1 A,
          2 B,
            3 C,
          2 D,
            3 C;
```

In the following example, a reference to A refers to the first A, A.A to the second A, and A.A.A to the third A.

```
declare 1 A,
          2 A,
            3 A;
```

In the following example, a reference to X refers to the first DECLARE statement. A reference to Y.Z is ambiguous. Y.Y.Z refers to the second Z, and Y.X.Z refers to the first Z.

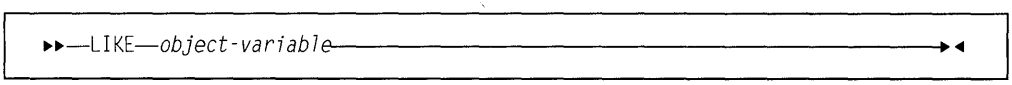
```
declare X;
declare 1 Y,
          2 X,
            3 Z,
            3 A,
          2 Y,
            3 Z,
            3 A;
```

For more information about name qualification, refer to "Scope of declarations" on page 132.

LIKE attribute

The LIKE attribute specifies that the name being declared has an organization that is logically the same as the referenced structure or union (object of the LIKE attribute). The object variable's member names and their attributes, including the dimension attribute, are effectively copied and become members of the name being declared. If necessary, the level numbers of the copied members are automatically adjusted. The object variable name and its attributes, including the dimension attribute, are ignored.

The syntax for the LIKE attribute is:



object-variable

can be a major structure, a minor structure, or a union. It must be known in the block containing the LIKE attribute specification. It can be qualified but must not be subscripted. The object or any of its members must not have the LIKE attribute or the REFER option.

The objects in all LIKE attributes are associated with declared names before any LIKE attributes are expanded.

New members cannot be added to the created structure or union. Any level number that immediately follows the object variable in the LIKE attribute must be equal to or less than the level number of the name with the LIKE attribute.

The following declarations yield the same structure for X.

```
dcl
  1 A(10) aligned static,
    2 B    bit(4),
    2 C    bit(4),
  1 X like A;

dcl
  1 X,
    2 B bit(4),
    2 C bit(4);
```

Notice that the dimension (DIM(10)), ALIGNED, and STATIC attributes are not copied as part of the LIKE expansion.

The LIKE attribute is expanded before the defaults are applied and before the ALIGNED and UNALIGNED attributes are applied to the contained elements of the LIKE object variable.

Examples

```
Declare 1 A,
          2 C,
            3 E(3) union,
              5 E1,
              5 E2,
            3 F;
Declare 1 B(10) union,
          2 C, 3 G, 3 H,
          2 D;
BEGIN;
Declare 1 C LIKE B;
Declare 1 D(2),
          5 BB LIKE A.C;
END;
```

Declarations C and D have the results shown in the following example.

```
dcl
    1 C,                /* DIM and UNION not copied. */
      2 C, 3 G, 3 H,
      2 D;

dcl 1 D(2),
      5 BB,
        6 E(3) union,   /* DIM(3) and UNION copied.      */
          7 E1,         /* Note adjusted level numbers.  */
          7 E2,
        6 F;
```

The following example is invalid because C.E has the LIKE attribute.

```
declare 1 A like C,
        1 B,
          2 C,
            3 D,
            3 E like X,
          2 F,
        1 X,
          2 Y,
          2 Z;
```

The following example is invalid because the LIKE attribute of A specifies a substructure, G.C, of a structure, G, declared with the LIKE attribute.

```
declare 1 A like G.C,
        1 B,
          2 C,
            3 D,
            3 E,
          2 F,
        1 G like B;
```

The following example is invalid because the LIKE attribute of A specifies a structure, C, within a structure, B, that contains a substructure, F, having the LIKE attribute.

```
declare 1 A like C,
        1 B,
          2 C,
            3 D,
            3 E,
          2 F like X,
        1 X,
          2 Y,
          2 Z;
```

Combinations of arrays, structures, and unions

Specifying the dimension attribute on a structure or union results in an *array of structures* or an *array of unions* respectively. The elements of such an array are structures or unions having identical names, levels, and members. For example, if a structure were used to hold meteorological data for each month of the years for the twentieth and the twenty-first centuries, it might be declared as follows:

```
Declare 1 Year(1901:2100),
          3 Month(12),
            5 Temperature,
              7 High decimal fixed(4,1),
              7 Low decimal fixed(4,1),
            5 Wind_velocity,
              7 High decimal fixed(3),
              7 Low decimal fixed(3),
           5 Precipitation,
             7 Total decimal fixed(3,1),
             7 Average decimal fixed(3,1),
          3 * char(0);
```

You could refer to the weather data for July 1991 by specifying Year(1991,7). Portions of this data could be referred to by Temperature(1991,7) and Wind_Velocity(1991,7). Precipitation.Total(1991,7) or Total(1991,7) would both refer to the total precipitation during July 1991.

Temperature.High(1991,3), which would refer to the high temperature in March 1991, is a subscripted qualified reference.

The need for subscripted qualified references becomes apparent when an array of structures or unions contains members that are arrays. In the following example, both A and B are structures.

```
declare 1 A (2,2),
          (2 B (2),
             3 C,
             3 D,
           2 E) fixed bin;
```

To refer to a data item, it may be necessary to use as many as three names and three subscripts. For example:

A(1,1).B	refers to B, an array of structures.
A(1,1)	refers to a structure.
A(1,1).B(1)	refers to a structure.
A(1,1).B(2).C	refers to an element.

As long as the order of subscripts remains unchanged, subscripts in such references can be moved to names at a lower or higher level. In the previous example, A.B.C(1,1,2) and A(1,1,2).B.C have the same meaning as A(1,1).B(2).C for the above array of structures. Unless all of the subscripts are moved to the lowest level, the reference is said to have *interleaved subscripts*, so A.B(1,1,2).C has interleaved subscripts.

Any item declared within an array of structures or unions inherits dimensions declared in the parent. In the previous declaration for the array of structures A, the array B is a three-dimensional structure, because it inherits the two dimensions declared for A. If B is unique and requires no qualification, any reference to a particular B would require three subscripts, two to identify the specific A and one to identify the specific B within that A.

Cross sections of arrays of structures or unions

A reference to a cross section of an array of structures or unions is not allowed. That is, the asterisk notation cannot be used in a reference unless all of the subscripts are asterisks.

Structure and union operations

Structures can be referenced in most contexts that any elementary variable can be referenced. For example, you can have structure references in assignments, I/O statements, and so on. References to unions or structures that contain unions, however, are limited to the following:

- Parameters and arguments
- Storage control and those built-in functions and subroutines that permit structures.

Structure and union mapping

Individual members of a union are mapped the same way as members of the structure. That is, each of the members, if not a union, is mapped as if it were a member of a structure. This means that the first storage location for each of the members of a union will not overlay each other if each of the members requires different alignment and therefore different padding before the beginning of the member.

Consider the following union:

```
dcl
  1 A union,
    2 B,
      3 C char(1),
      3 D fixed bin(31),
    2 E,
      3 F char(2),
      3 G fixed bin(31);
```

Three bytes of padding are added between A and B. Two bytes are added between A and E.

In order to ensure that the first storage location of each of the members of a union is the same, make sure that the first member of each has the same alignment requirement and it is the same as the highest alignment of any of its members (or its member's members).

The remainder of the discussion applies to members of a structure or union, which may be minor structures or elementary variables.

For any major or minor structure, the length, alignment requirement, and position relative to an 8-byte boundary depend on the lengths, alignment requirements, and relative positions of its members. The process of determining these requirements for each level and for the complete structure is known as *structure mapping*.

You can use structure mapping for determining the record length required for a structure when record-oriented input/output is used, and determining the amount of padding or rearrangement required for correct alignment of a structure for locate-mode input/output.

The structure mapping process minimizes the amount of unused storage (padding) between members of the structure. It completes the entire process before the structure is allocated, according (in effect) to the rules discussed in the following paragraphs.

Structure mapping is not a physical process. Terms such as *shifted* and *offset* are used purely for ease of discussion, and do not imply actual movement in storage. When the structure is allocated, the relative locations are already known as a result of the mapping process.

The mapping for a complete structure reduces to successively combining pairs of items (elements, or minor structures whose individual mappings have already been determined). Once a pair has been combined, it becomes a unit to be paired with another unit, and so on until the complete structure is mapped. The rules for the process are categorized as:

- Rules for determining the order of pairing
- Rules for mapping one pair.

These rules are described below, and an example shows an application of the rules in detail. It is necessary to understand the difference between the *logical level* and the *level number* of structure elements. The logical levels are immediately apparent if the structure declaration is written with consistent level numbers or suitable indentation (as in the detailed example given after the rules). In any case, you can determine the logical level of each item in the structure by applying the fol-

lowing rule to each item in turn, starting at the beginning of the structure declaration:

Note: The logical level of a given item is always one unit deeper than that of its immediate containing structure.

In the following example, the lower line shows the logical level for each item in the declaration.

```
dcl 1 A, 4 B, 5 C, 5 D, 3 E, 8 F, 7 G;

    1   2   3   3   2   3   3
```

Rules for order of pairing

The steps in determining the order of pairing are as follows:

1. Find the minor structure at the deepest logical level (which we will call logical level n).
2. If more than one minor structure has the logical level n, take the first one that appears in the declaration.
3. Pair the first two elements appearing in this minor structure, thus forming a unit. Use the rules for mapping one pair. (See "Rules for mapping one pair.")
4. Pair this unit with the next element (if any) declared in the minor structure, thus forming a larger unit.
5. Repeat step 4 until all the elements in the minor structure have been combined into one unit. This completes the mapping for this minor structure; its alignment requirement and length, including any padding, are now determined and will not change (unless you change the structure declaration). Its offset from a doubleword boundary is also now determined; note that this offset will be significant during mapping of any containing structure, and it may change as a result of such mapping.
6. Repeat steps 3 through 5 for the next minor structure (if any) appearing at logical level n in the declaration.
7. Repeat step 6 until all minor structures at logical level n have been mapped. Each of these minor structures can now be thought of as an element for structure mapping purposes.
8. Repeat the pairing process for minor structures at the next higher logical level; that is, make n equal to (n-1) and repeat steps 2 through 7.
9. Repeat step 8 until n = 1; then repeat steps 3 through 5 for the major structure.

Rules for mapping one pair

For purposes of this explanation, think of storage as contiguous doublewords, each having 8 bytes, numbered 0 through 7, which indicate the offset from a doubleword boundary. Think of the bytes as numbered continuously from 0 onwards, starting at any byte, so that lengths and offsets from the start of the structure can be calculated.

1. Begin the first element of the pair on a doubleword boundary; or, if the element is a minor structure that has already been mapped, offset it from the doubleword boundary by the amount indicated.
2. Begin the second element of the pair at the first valid position following the end of the first element. This position will depend on the alignment requirement of

the second element. (If the second element is a minor structure, its alignment requirement will have been determined already.)

3. Shift the first element towards the second element as far as the alignment requirement of the first allows. The amount of shift determines the offset of this pair from a doubleword boundary.

After this process has been completed, any padding between the two elements has been minimized and will not change throughout the rest of the operation. The pair is now a unit of fixed length and alignment requirement; its length is the sum of the two lengths plus padding, and its alignment requirement is the higher of the two alignment requirements (if they differ).

Effect of UNALIGNED attribute

The example of structure mapping given below shows the rules applied to a structure declared ALIGNED. Mapping of aligned structures is more complex because of the number of alignment requirements. The effect of the UNALIGNED attribute is to reduce to 1 byte the alignment requirements for halfwords, fullwords, and doublewords, and to reduce to 1 bit the alignment requirement for bit strings. The same structure mapping rules apply, but the reduced alignment requirements are used. The only unused storage will be bit padding within a byte when the structure contains bit strings.

AREA data cannot be unaligned.

If a structure has the UNALIGNED attribute and it contains an element that cannot be unaligned, UNALIGNED is ignored for that element; the element is aligned and an error message is put out. For example, in a program with the following declaration, C is given the attribute ALIGNED because the inherited attribute UNALIGNED conflicts with AREA.

```
DECLARE 1 A UNALIGNED,
          2 B,
          2 C AREA(100);
```

Example of structure mapping

The following example shows the application of the structure mapping rules for a structure with the specified declaration.

```
declare 1 A aligned,
          2 B fixed bin(31),
          2 C,
            3 D float decimal(14),
            3 E,
              4 G,
                5 H character(2),
                5 I float decimal(13),
              4 J fixed binary(31,0),
            3 K character(2),
            3 L fixed binary(20,0),
          2 M,
            3 N,
              4 P fixed binary(15),
              4 Q character(5),
              4 R float decimal(2),
            3 S,
              4 T float decimal(15),
              4 U bit(3),
              4 V char(1),
            3 W fixed bin(31),
          2 X picture '$9V99';
```

The minor structure at the deepest logical level is G, so this is mapped first. Then E is mapped, followed by N, S, C, and M, in that order.

For each minor structure, a table in Figure 34 shows the steps in the process, and a diagram in Figure 35 shows a visual interpretation of the process. Finally, the major structure A is mapped as shown in Figure 36. At the end of the example, the structure map for A is set out in the form of a table (Figure 37) showing the offset of each member from the start of A.

	Name of Element	Alignment Requirement	Length	Offset from Doubleword Begin	Offset from Doubleword End	Length of Padding	Offset from Minor Structure
Step 1	H	Byte	2	0	1		
	I	Doubleword	8	0	7		
Step 2	*H	Byte	2	6	7		0
	I	Doubleword	8	0	7	0	2
Minor Structure	G	Doubleword	10	6	7		
Step 1	F	Fullword	8	0	7		
	G	Doubleword	10	6	7		
Step 2	*F	Fullword	8	4	3		0
	G	Doubleword	10	6	7	2	10
Step 3	F & G	Doubleword	20	4	7		
	J	Fullword	4	0	3	0	20
Minor Structure	E	Doubleword	24	4	3		
Step 1	P	Halfword	2	0	1		0
	Q	Byte	5	2	6		2
Step 2	P & Q	Halfword	7	0	6		
	R	Fullword	4	0	3	1	8
Minor Structure	N	Fullword	12	0	3		
Step 1	T	Doubleword	8	0	7		0
	U	Byte	1	0	0	0	8
Step 2	T & U	Doubleword	9	0	0		
	V	Byte	1	1	1	0	9
Minor Structure	S	Doubleword	10	0	1		
Step 1	D	Doubleword	8	0	7		0
	E	Doubleword	24	4	3	4	12
Step 2	D & E	Doubleword	36	0	3		
	K	Byte	2	4	5	0	36
Step 3	D, E, & K	Doubleword	38	0	5		
	L	Fullword	4	0	3	2	40
Minor Structure	C	Doubleword	44	0	3		
Step 1	N	Fullword	12	0	3		
	S	Doubleword	10	0	1		
Step 2	*N	Fullword	12	4	7		0
	S	Doubleword	10	0	1	0	12
Step 3	N & S	Doubleword	22	4	1		
	W	Fullword	4	4	7	2	24
Minor Structure	M	Doubleword	28	4	7		

*First item shifted right

Figure 34. Mapping of example structure

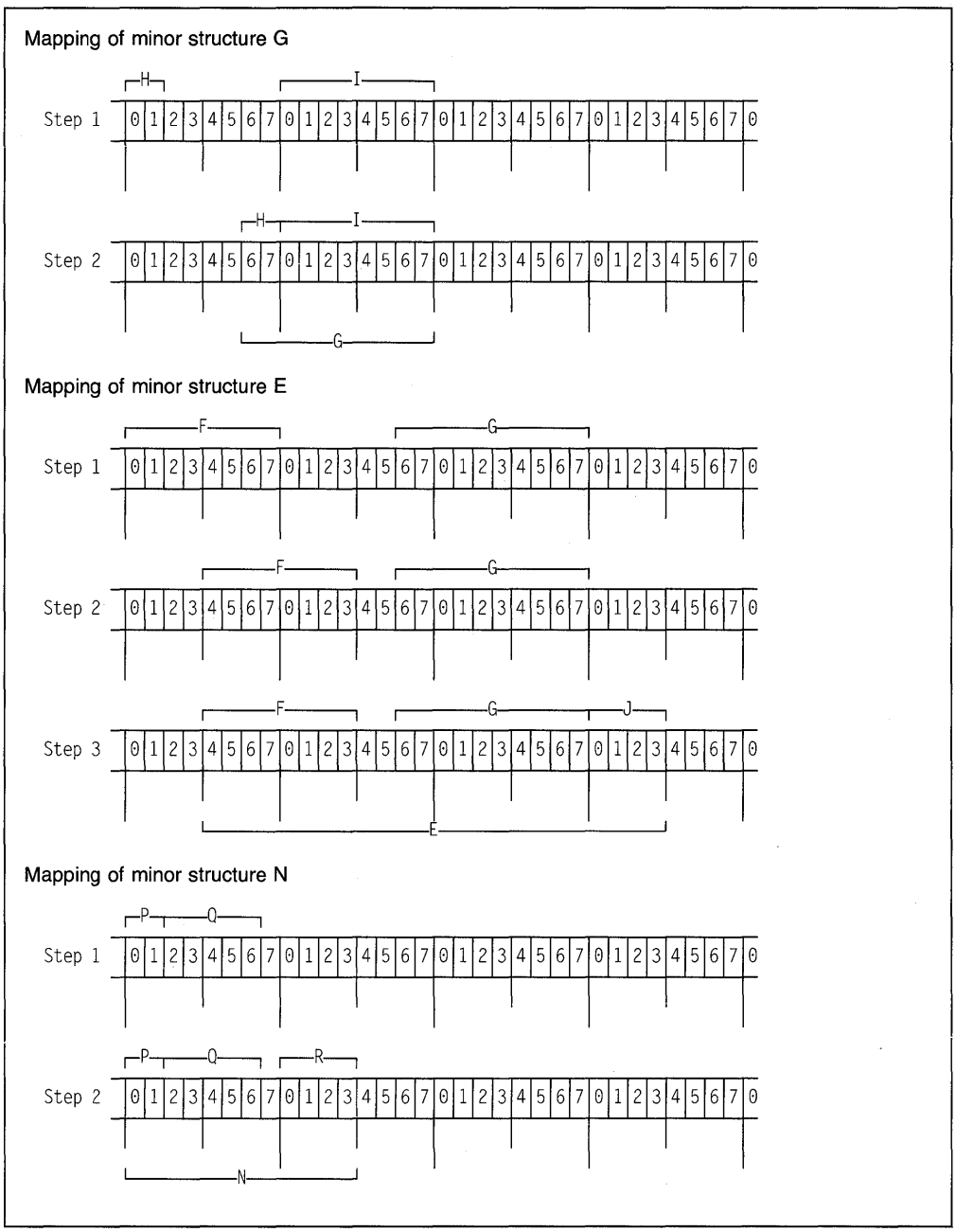


Figure 35 (Part 1 of 2). Mapping of minor structures

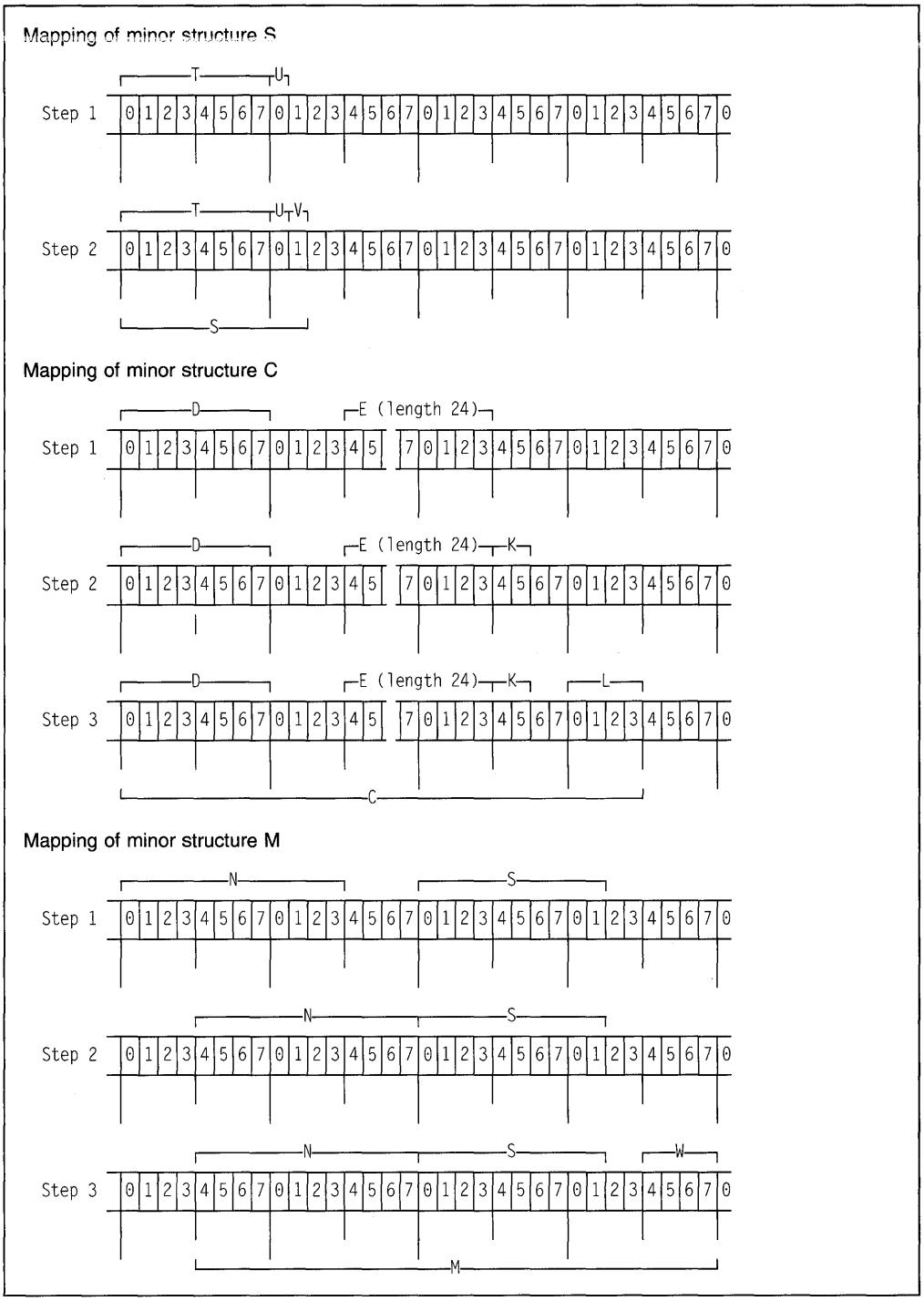


Figure 35 (Part 2 of 2). Mapping of minor structures

Structure mapping example

	Name of Item	Alignment Required	Length	Offset from Doubleword Begin	Offset from Doubleword End	Length of Padding	Offset from A
Step 1	B	Fullword	4	0	3		
	C	Doubleword	44	0	3		
Step 2	B*	Fullword	4	4	7		0
	C	Doubleword	44	0	3	0	4
Step 3	B & C	Doubleword	48	4	3		
	M	Doubleword	28	4	7	0	48
Step 4	B, C, & M	Doubleword	76	4	7		
	X	Byte	4	0	3	0	76
	A	Doubleword	80	4	3		

* First item shifted right

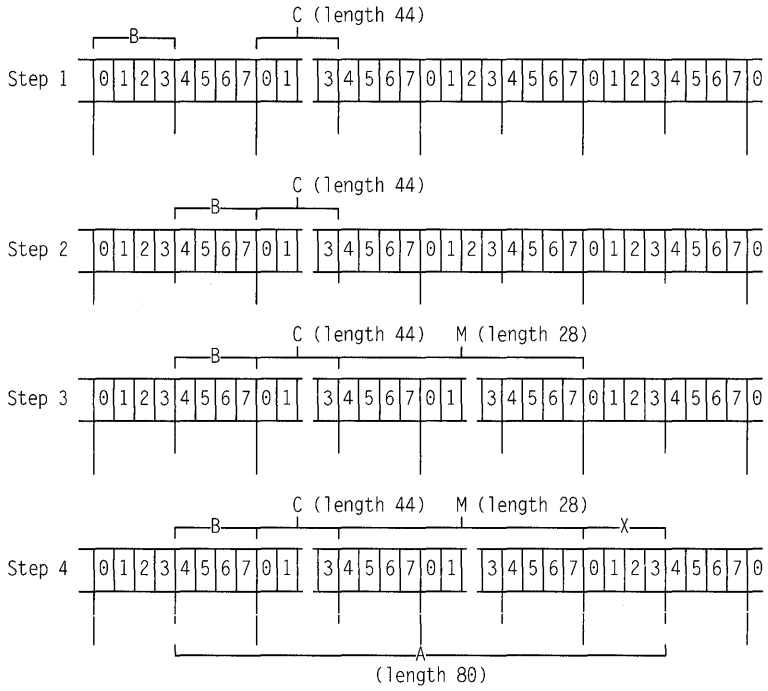


Figure 36. Mapping of major structure A

A				From A
B				0
C			From C	4
D			0	4
padding (4)			8	12
E		From E	12	16
F		0	12	16
padding (2)		8	20	24
G	From G	10	22	26
H	0	10	22	26
I	2	12	24	28
J		20	32	36
K			36	40
padding (2)			38	42
L			40	44
M			From M	48
N		From N	0	48
P		0	0	48
Q		2	2	50
padding (1)		7	7	55
R		8	8	56
S		From S	12	60
T		0	12	60
U		8	20	68
V		9	21	69
padding (2)			22	70
W			24	72
X				76

Figure 37. Offsets in final mapping of structure A

Chapter 8. Statements

Chapter 8. Statements

This chapter lists all of the PL/I and macro facility statements. Statements that are described in other chapters are listed with cross-references to the full descriptions.

%ACTIVATE statement

The %ACTIVATE macro facility statement is described in "%ACTIVATE" on page 453.

ALLOCATE statement

The ALLOCATE statement is described in Chapter 9, "Storage control" on page 198.

Assignment statement

The assignment statement evaluates an expression and assigns its value to one or more target variables.

This statement is used for internal data movement, as well as for specifying computations. (The GET and PUT statements with the STRING option can also be used for internal data movement. Additionally, the PUT statement can specify computations to be done. See Chapter 12, "Stream-oriented data transmission.")

Because the attributes of the target variable or pseudovariable can differ from the attributes of the source (a variable, a constant, or the result of an expression), the assignment statement might require conversions (see Chapter 5, "Data conversion").

The syntax for the assignment statement is:

Area assignment is described in "Area data and attribute" on page 211.

The remaining text discusses:

- The requirements for target variables
- How element and aggregate assignments are performed
- How BY NAME assignments are performed
- How multiple assignment are performed.

Examples of assignments begin on page 170.

Target variables

The target variables can be element, array, or structure variables; or pseudovariables.

Array targets

For array assignments, each target variable must be an array of scalars or structures. The source must be a scalar or an expression with the same number of dimensions and the same bounds for all dimensions as for the target.

Union targets

Union assignments are not permitted.

Structure targets

For structure assignments, each target variable must be a structure. The right-hand side can be a structure reference or element expression.

If the target in an assignment statement without the BYNAME option is a structure, the source must be either a scalar or a structure with the same structuring as the target structure. That is, the source must have the same number of members. Members that are structures must have the same number of members and the same dimensions.

How assignments are performed

Element assignments

Element assignments are performed as follows:

1. First to be evaluated are subscripts, POSITION attribute expressions, locator qualifications of the target variables, and the second and third arguments of SUBSTR pseudovariable references.
2. The expression on the right-hand side is then evaluated.
3. For each target variable (in left to right order), the expression is converted to the characteristics of the target variable according to the rules for data conversion. The converted value is then assigned to the target variable.

Aggregate assignments

Aggregate assignments (array and structure assignments) are expanded into a series of element assignments as follows:

1. The label prefix of the original statement is applied to a null statement preceding the other generated statements.
2. Array and structure assignments, when there are more than one, are done iteratively.
3. Any assignment statement can be generated by a previous array or structure assignment. The first target variable in an aggregate assignment is known as the master variable. (It can also be the first argument of a pseudovariable). If the master variable is an array, an array expansion is performed; otherwise, a structure expansion is performed.
4. If an aggregate assignment meets a certain set of conditions, it can be done as a whole instead of being expanded into a series of element assignments. Two

conditions are if the arrays are not interleaved, or if the structures are contiguous and have the same format.

In array assignments, all array operands must have the same number of dimensions and identical bounds. The array assignment is expanded into a loop as follows:

```
do J1 = lbound(Master-variable,1) to
        hbound(Master-variable,1);
do J2 = lbound(Master-variable,2) to
        hbound(Master-variable,2);
  ⋮
do jn = lbound(Master-variable,n) to
        hbound(Master-variable,n);

generated assignment statement

end;
```

In this expansion, `n` is the number of dimensions of the master variable that are to participate in the assignment. In the generated assignment statement, all array operands are fully subscripted, using (from left to right) the dummy variables `j1` to `jn`. If an array operand appears with no subscripts, it will only have the subscripts `j1` to `jn`. If cross-section notation is used, the asterisks are replaced by `j1` to `jn`. If the original assignment statement has a condition prefix, the generated assignment statement is given this condition prefix.

If the generated assignment statement is a structure assignment, it is expanded as described next.

In structure assignments where the BY NAME option is not specified:

- None of the operands can be arrays, although they can be structures that contain arrays.
- All of the structure operands must have the same number, *k*, of immediately contained items.
- The assignment statement is replaced by *k* generated assignment statements.
  - The *i*th generated assignment statement is derived from the original assignment statement by replacing each structure operand by its *i*th contained item; such generated assignment statements can require further expansion.
  - All generated assignment statements are given the condition prefix of the original statement.

In structure assignments where the BY NAME option is given, the structure assignment is expanded according to the steps below, which can generate further array and structure assignments. None of the operands can be arrays.

1. The first item immediately contained in the master variable is considered.
2. If each structure operand and target variable has an immediately contained item with the same name, an assignment statement is generated as follows:
   a. The statement is derived by replacing each structure operand and target variable with its immediately contained item that has this name. If any structure contains no such name, no statement is generated.
   b. If the generated assignment is a structure or array-of-structures assignment, BY NAME is appended.

c. All generated assignment statements are given the condition prefix of the original assignment statement.

d. A target structure must not contain unions.

3. Step 2 is repeated for each of the items immediately contained in the master variable. The assignments are generated in the order of the items contained in the master variable.

Multiple assignments

Assignments can be made to multiple variables in a single assignment statement. For example:

```
A,X = B + C;
```

The value of `B + C` is assigned to both `A` and `X`. In general, it has the same effect as the following statements:

```
Temporary = B + C;
A = Temporary;
X = Temporary;
```

The source in the assignment statement must be scalar. If the source is a constant, it is assigned to each of the targets from left to right. If the source is not a constant, it is assigned to a temporary variable, which is then assigned to each of the targets from left to right.

The target can be any reference permitted in a simple assignment.

BY NAME is not permitted in multiple assignments.

Example of moving internal data

The following example of the assignment statement can be used for internal data movement. The value of the expression on the right of the assignment symbol is to be assigned to the variable on the left.

```
NTOT=TOT;
```

Example of assigning expression values

The following example includes an expression whose value is to be assigned to the variable on the left of the assignment symbol:

```
Av=(Av*Num+Tav*Tnum)/(Num+Tnum);
```

Example of assigning a structure using BY NAME

The following example illustrates structure assignment using the BY NAME option:

```
declare      declare      declare
1 one,       1 two,       1 three,
 2 Part1,     2 Part1,     2 Part1,
  3 Red,       3 Blue,      3 Red,
  3 Orange,    3 Green,     3 Blue,
 2 Part2,     3 Red,       3 Brown,
  3 Yellow,  2 Part2,     2 Part2,
  3 Blue,      3 Brown,     3 Yellow,
  3 Green;     3 Yellow;    3 Green;

|1|   One = Two, by name;
|2|   One.Part1 = Three.Part1, by name;
```

|1| The first assignment statement is the same as the following:

```
One.Part1.Red    = Two.Part1.Red;
One.Part2.Yellow = Two.Part2.Yellow;
```

|2| The second assignment statement is the same as the following:

```
One.Part1.Red = Three.Part1.Red;
```

%assignment statement

The %assignment macro facility statement is discussed in "%assignment" on page 454.

BEGIN statement

The BEGIN statement is described in Chapter 6, "Program organization" on page 89.

CALL statement

The CALL statement is described in "CALL statement" on page 120.

CLOSE statement

The CLOSE statement is described in Chapter 10, "Input and output" on page 228.

%DEACTIVATE statement

The DEACTIVATE statement is described in "%DEACTIVATE" on page 454.

DECLARE statement

The DECLARE statement is described in "DECLARE statement" on page 129.

%DECLARE statement

The %DECLARE macro facility statement is discussed in "%DECLARE" on page 455.

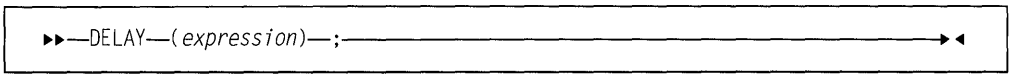
DEFAULT statement

The DEFAULT statement is described in "DEFAULT statement" on page 142.

DELAY statement

The DELAY statement suspends the execution of the next statement in the application program for the specified period of time.

The syntax for the DELAY statement is:



expression
specifies an expression that is evaluated and converted to FIXED BIN(M,0). Execution is suspended for the number of milliseconds specified.

The maximum wait time is 23 hours and 59 minutes.

For example:

```
delay (20);
```

suspends execution for 20 milliseconds.

```
delay (10**3);
```

suspends execution for 1 second.

```
delay (10*10**3);
```

suspends execution for 10 seconds.

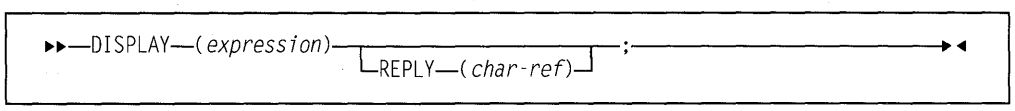
DELETE statement

The DELETE statement is described in "DELETE statement" on page 245.

DISPLAY statement

The DISPLAY statement displays a message on the user's screen and optionally requests the user to enter a response to the message.

The syntax for the DISPLAY statement is:



expression
is converted, where necessary, to a character string. This character string is displayed. It can contain mixed character data. If the expression has the GRAPHIC attribute, it is not converted.

REPLY (char-ref)
specifies a character variable that receives the user entered response. You cannot use pseudovariables. The response can contain CHARACTER, GRAPHIC, or mixed data.

The REPLY option will suspend program execution until the user enters a response. In some environments (for example, OS/2 Presentation Manager), the DISPLAY statement, even without the REPLY option, suspends execution of the application until the user acknowledges the message.

If GRAPHIC data is entered in the REPLY, it is received as character data that contains mixed data. Such character data can be converted to GRAPHIC data using the GRAPHIC BUILTIN.

Example:

```
display ('Communication link established.');
```

displays the message

```
Communication link established.
```

DO statement

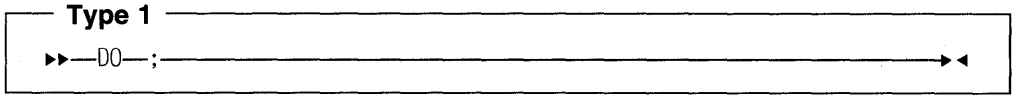
The DO statement and its corresponding END statement, delimit a group of statements collectively called a do-group.

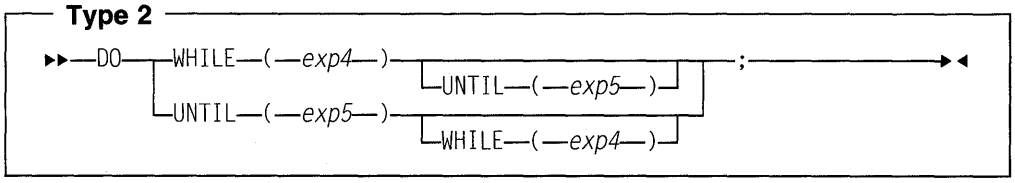
Note: Condition prefixes are invalid on DO statements.

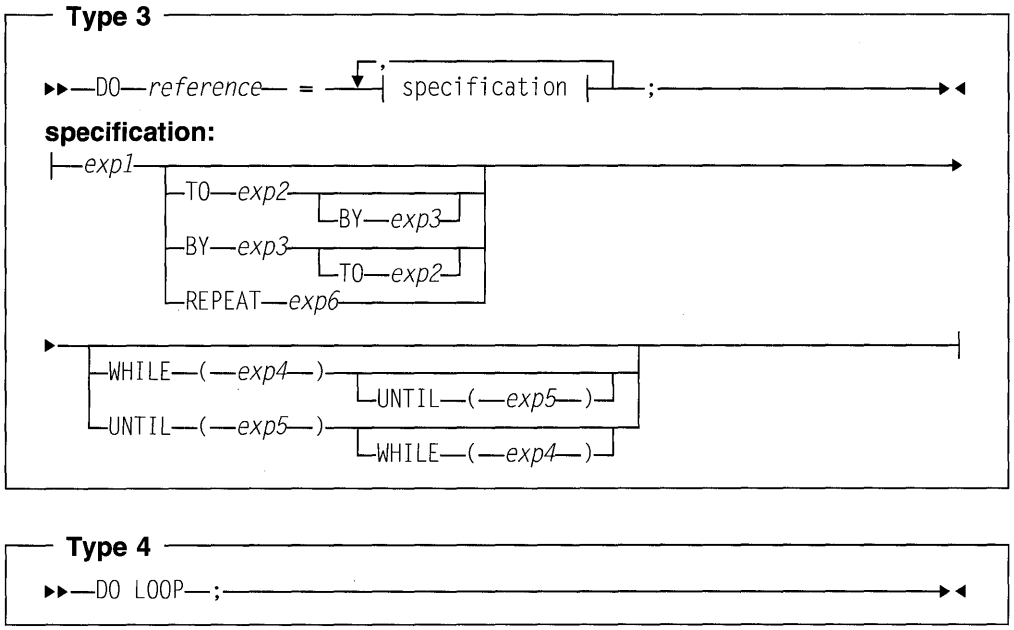
The syntax for the DO statement is:

Notes:

1. exp*n* is an abbreviation for expression *n*.
2. Condition prefixes are invalid on DO statements.







expn
an abbreviation for *expression n*

Type 1

The type 1 do-group specifies that the statements in the group are executed. It does not provide for the repetitive execution of the statements within the group.

Types 2 and 3

Types 2 and 3 provide for the repetitive execution of the statements within the do-group.

WHILE (exp4)
specifies that, before each repetition of do-group, *exp4* is evaluated and, if necessary, converted to a bit string. If any bit in the resulting string is 1, the do-group is executed. If all bits are 0, or the string is null, execution of the Type 2 do-group is terminated. For Type 3, only the execution associated with the specification containing the WHILE option is terminated. Execution for the next specification, if one exists, then begins.

UNTIL (exp5)
specifies that, after each repetition of do-group, *exp5* is evaluated, and, if necessary, converted to a bit string. If all the bits in the resulting string are 0, or the string is null, the next iteration of the do-group is executed. If any bit is 1, execution of the Type 2 do-group is terminated. For Type 3, only the execution associated with the specification containing the UNTIL option is terminated. Execution for the next specification, if one exists, then begins.

reference
Pseudovariables cannot be used as a reference. All data types are allowed.

The generation, g, of a reference is established once at the beginning of the do-group, immediately before the initial value expression (exp1) is

evaluated. If the reference generation is changed to h in the do-group, the do-group continues to execute with the reference derived from the generation g. However, any reference to the reference inside the do-group is a reference to generation h. It is an error to free generation g in the do-group.

If a reference is made to a reference after the last iteration is completed, the value of the variable is the value that was out of range of the limit set in the specification. That is, if:

- The BY value is positive and the reference is greater than the TO value
- The BY value is negative and the reference is less than the TO value

of the limit set in the specification.

If reference is a program-control data variable other than a locator, the BY and TO options cannot be used in specification.

exp1 specifies the initial value of the reference.

If *TO*, *BY*, and *REPEAT* are all omitted from a specification, there is a single execution of the do-group, with the reference having the value of *exp1*. If *WHILE(exp4)* is included, the single execution does not take place unless *exp4* is true.

TO exp2

exp2 is evaluated at entry to the specification and saved. This saved value specifies the terminating value of the reference. Execution of the statements in a do-group terminates for a specification as soon as the value of the reference, when tested at the end of the do-group, is out of range. Execution of the next specification, if one exists, then begins.

If *TO exp2* is omitted from a specification, and if *BY exp3* is specified, repetitive execution continues until it is terminated by the WHILE or UNTIL option, or until another statement transfers control out of the do-group.

BY exp3

exp3 is evaluated at entry to the specification and saved. This saved value specifies the increment to be added to the reference after each execution of the do-group.

If *BY exp3* is omitted from a specification, and if *TO exp2* is specified, *exp3* defaults to 1.

If BY 0 is specified, the execution of the do-group continues indefinitely unless it is halted by a WHILE or UNTIL option, or control is transferred to a point outside the do-group.

REPEAT exp6

exp6 is evaluated and assigned to the reference after each execution of the do-group. Repetitive execution continues until it is terminated by the WHILE or UNTIL option, or another statement transfers control out of the do-group.

The following sections give more information about using Type 2 and Type 3 DO groups. Examples of DO groups begin on page 179.

Using type 2 WHILE and UNTIL

If a Type 2 DO specification includes both the WHILE and UNTIL option, the DO statement provides for repetitive execution as defined by the following:

```
Label:  do while (exp4)
           until (exp5)
           statement-1
           ⋮
           statement-n
           end;
Next:   statement /* Statement following the do group */
```

The above is equivalent to the following expansion:

```
Label:  if (exp4) then;
        else
          go to Next;
         statement-1
        ⋮
        statement-n
Label2: if (exp5) then;
         else
 go to Label;
Next:   statement /* Statement following the do group */
```

If the WHILE option is omitted, the IF statement at label `Label` is replaced by a null statemen Note that if the WHILE option is omitted, statements 1 through n are executed at least once.

If the UNTIL option is omitted, the IF statement at label `Label2` in the expansion is replaced by the statement GO TO Label.

Using type 3 with one specification

The following sequence of events summarizes the effect of executing a do-group with one specification:

1. If reference is specified and BY and TO options are also specified, *exp1*, *exp2*, and *exp3* will be evaluated prior to the assignment of *exp1* to the reference. Then the initial value is assigned to reference. For example:

   ```
   do reference = exp1 to exp2 by exp3;
   ```

   For a variable that is not a pseudovariable, the above action of the do-group definition is equivalent to the following expansion:

   ```
   e1=exp1;
   e2=exp2;
   e3=exp3;
   V=E1;
   ```

   The variable `v` is a compiler-created based variable with the same attributes as the reference. `e1`, `e2`, and `e3` are compiler-created variables.

2. If the TO option is present, test the value of the control variable against the previously-evaluated expression `(e2)` in the TO option.
3. If the WHILE option is specified, evaluate the expression in the WHILE option. If it is *false*, leave the do-group.
4. Execute the statements in the do-group.

5. If the UNTIL option is specified, evaluate the expression in the UNTIL option. If it is *true*, leave the do-group.
6. If there is a reference:
   a. If the TO or BY option is specified, add the previously-evaluated *exp3* (e3) to the reference.
   b. If the REPEAT option is specified, evaluate the *exp6* and assign it to the reference.
   c. If the TO, BY, and REPEAT options are all absent, leave the do-group.
7. Go to 2 on page 176.

Using type 3 with two or more specifications

If the DO statement contains more than one specification, the second expansion is analogous to the first expansion in every respect. However, the statements in the do-group are not actually duplicated in the program. A succeeding specification is executed only after the preceding specification has been terminated.

When execution of the last specification terminates, control passes to the statement following the do-group.

Control can transfer into a do-group from outside the do-group only if the do-group is delimited by the DO statement in Type 1. Control can also return to a do-group from a procedure or ON-unit invoked from within that do-group.

Using type 3 with TO, BY, REPEAT

The TO and BY options let you vary the reference in fixed positive or negative increments. In contrast, the REPEAT option, which is an alternative to the TO and BY options, lets you vary the control variable nonlinearly. The REPEAT option can also be used for nonarithmetic control variables (such as pointer).

If the Type 3 DO specification includes the TO and BY options, the action of the do-group is defined by the following:

```
Label:  do variable=
            exp1
            to exp2
            by exp3
            while (exp4)
            until(exp5);
            statement-1
            :
            statement-m
Label1:  end;
Next:    statement
```

The action of the previous do-group definition is equivalent to the following expansion. In this expansion, v is a compiler-created variable with the same attributes as variable; and e1, e2, and e3 are compiler-created variables:

```
Label:  e1=exp1;
        e2=exp2;
        e3=exp3;
        V=e1;
Label2: if (e3>=0)&(v>e2)|(e3<0)&(v<e2) then
          go to Next;
        if (exp4) then;
          else
           go to Next;
          statement-1
          :
          statement-m
Label1: if (exp5) then
          go to Next;
Label3: V=V+e3;
        go to Label2;
Next:   statement
```

If the specification includes the REPEAT option, the action of the do-group is defined by the following:

```
Label:  do variable=
            exp1
            repeat exp6
            while (exp4)
            until(exp5);
            statement-1
            :
            statement-m
Label1:  end;
Next:    statement
```

The action of the previous do-group definition is equivalent to the following expansion:

```
Label:  e1=exp1;
        V=e1;
Label2: ;
        if (exp4) then;
        else
          go to Next;
        statement-1
        :
        statement-m
Label1: if (exp5) then
           go to Next;
Label3: V=exp6;
           go to Label2;
Next:   statement
```

Additional rules for the sample expansions are as follows:

1. The previous expansion only shows the result of one specification. If the DO statement contains more than one specification, the statement labeled NEXT is the first statement in the expansion for the next specification. The second expansion is analogous to the first expansion in every respect. Note, however, that statements 1 through m are not actually duplicated in the program.
2. If the WHILE clause is omitted, the IF statement immediately preceding statement-1 in each of the expansions is also omitted.
3. If the UNTIL clause is omitted, the IF statement immediately following statement-m in each of the expansions is also omitted.

Type 4

LOOP specifies infinite iteration. FOREVER is a synonym of LOOP.

The only way to exit this loop is by a LEAVE or GO TO, or by terminating a procedure or the program.

For example:

```
dcl payroll file;
dcl 1 payrec,
     2 type char,
     2 subtype char,
     2 * char(100);

Readfile:
  Do loop;

    Read file(payroll) into(payrec);

    If payrec.type = 'E'
      then leave; /* like goto After_ReadFile */

    If payrec.type = '1' then
      Do;
           /* process first part of record */

        If payrec.subtype = 'S'
          then iterate Readfile; /* like goto End_ReadFile */

           /* process remainder of record */
      End;

End_ReadFile:
  End;
After_ReadFile:;
```

Examples of basic repetitions

In the following example, the do-group is executed ten times, while the value of the reference `I` progresses from 1 through 10.

```
do I = 1 to 10;
:
end;
```

The effect of this DO and END statement is equivalent to:

```
   I = 1;
A: if I > 10 then go to B;
⋮
   I = I +1;
   go to A;
B: next statement
```

The following DO statement executes the do-group three times: once for each assignment of 'Tom', 'Dick', and 'Harry' to Name.

```
do Name = 'Tom', 'Dick', 'Harry';
```

The following statement specifies that the do-group executes thirteen times: ten times with the value of I equal to 1 through 10, and three times with the value of I equal to 13 through 15:

```
do I = 1 to 10, 13 to 15;
```

Repetition using the reference as a subscript

The reference of a DO statement can be used as a subscript in statements within the do-group, so that each execution deals with successive elements of a table or array.

In the following example, the first ten elements of A are set to 1 through 10 in sequence:

```
do I = 1 to 10;
   A(I) = I;
end;
```

Repetition with TO and BY

The following example specifies that the do-group is executed five times, with the value of I equal to 2, 4, 6, 8, and 10:

```
do I = 2 to 10 by 2;
```

If negative increments of the reference are required, the BY option must be used. For example, the following is executed with I equal to 10, 8, 6, 4, 2, 0, and -2:

```
do I = 10 to -2 by -2;
```

In the following example, the do-group is executed with I equal to 1, 3, 5:

```
I=2;
do I=1 to I+3 by I;
⋮
end;
```

It is equivalent to the following:

```
do I=1 to 5 by 2;
⋮
end;
```

Example of DO with WHILE, UNTIL

The WHILE and UNTIL options make successive executions of the do-group dependent upon a specified condition. For example:

```
do while (A=B);
⋮
end;
```

is equivalent to the following:

```
S:  if A=B then;
       else goto R;
       ⋮
       goto S;
R:  next statement
```

The example:

```
do until (A=B);
⋮
end;
```

is equivalent to the following:

```
S:
   ⋮
   if (A=B) then goto R;
   goto S;
R:  next statement
```

In the absence of other options, a do-group headed by a DO UNTIL statement is executed at least once, but a do-group headed by a DO WHILE statement might not be executed at all. That is, the statements `DO WHILE (A=B)` and `DO UNTIL (A¬=B)` are not equivalent.

In the following example, if `A¬=B` when the DO statement is first encountered, the do-group is not executed at all.

```
do while(A=B) until(X=10);
```

However, if `A=B`, the do-group is executed. If `X=10` after an execution of the do-group, no further executions are performed. Otherwise, a further execution is performed provided that `A` is still equal to `B`.

In the following example, the do-group is executed at least once, with `I` equal to 1:

```
do I=1 to 10 until(Y=1);
```

If `Y=1` after an execution of the do-group, no further executions are performed. Otherwise, the default increment (BY 1) is added to `I`, and the new value of `I` is compared with 10. If `I` is greater than 10, no further executions are performed. Otherwise, a new execution commences.

The following statement specifies that the do-group executes ten times while `C(I)` is less than zero, and then (provided that `A` is equal to `B`) once more:

```
do I = 1 to 10 while (C(I)<0),
       11 while (A = B);
```

Example of REPEAT

In the following example, the do-group is executed with I equal to 1, 2, 4, 8, 16, and so on:

```
do I = 1 repeat 2*I;
⋮
end;
```

This is equivalent to the following:

```
    I=1;
A:
    ⋮
    I=2*I;
    goto A;
```

In the following example, the first execution of the do-group is performed with I=1.

```
do I=1 repeat 2*I until(I=256);
```

After this and each subsequent execution of the do-group, the UNTIL expression is tested. If I=256, no further executions are performed. Otherwise, the REPEAT expression is evaluated and assigned to I, and a new execution commences.

The following example shows a DO statement used to locate a specific item in a chained list:

```
do P=Phead repeat P -> Fwd
     while(P¬=null())
     until(P->Id=Id_to_be_found);
end;
```

The value Phead is assigned to P for the first execution of the do-group. Before each subsequent execution, the value P -> Fwd is assigned to P. The value of P is tested before the first and each subsequent execution of the do-group. If it is null, no further executions are performed.

The following statement specifies that the do-group is to be executed nine times, with the value of I equal to 1 through 9; then successively with the value of I equal to 10, 20, 40, and so on. Execution ceases when the do-group has been executed with a value of I greater than 10000.

```
do I = 1 to 9, 10 repeat 2*I
               until (I>10000);
⋮
end;
```

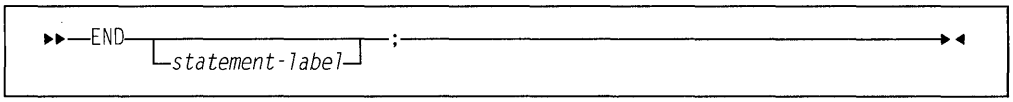
%DO statement

The %DO macro facility statement is discussed in "%DO" on page 455.

END statement

The END statement ends a DO, SELECT, PACKAGE, BEGIN or PROCEDURE. Every block or group must have an END statement.

The syntax for the END statement is:



statement-label

cannot be subscripted. If a statement-label follows END, the END statement closes the unclosed group or block headed by the nearest preceding DO, SELECT, PACKAGE, BEGIN, or PROCEDURE statement having that statement-label. Every block with a DO, SELECT, PACKAGE, BEGIN or PROCEDURE statement must have a corresponding END statement.

If a statement-label does not follow END, the END statement closes the one group or block headed by the nearest preceding DO, SELECT, PACKAGE, BEGIN, or PROCEDURE statement for which there is no other corresponding END statement.

Execution of a block terminates when control reaches the END statement for the block. However, it is not the only means of terminating a block's execution, even though each block must have an END statement. (See Chapter 6, "Program organization" on page 89 for more details.)

If control reaches an END statement for a procedure, it is treated as a RETURN statement.

Normal termination of a program occurs when control reaches the END statement of the main procedure.

%END statement

The %END macro facility statement is discussed in "%END" on page 456.

EXIT statement

The EXIT statement is identical to STOP. Refer to "STOP statement" on page 195.

FETCH statement

The FETCH statement is described in "FETCH statement" on page 103.

FORMAT statement

The FORMAT statement is described in Chapter 12, "Stream-oriented data transmission" on page 252.

FREE statement

The FREE statement is described in Chapter 9, "Storage control" on page 198.

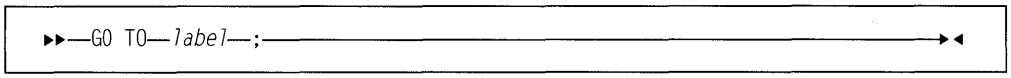
GET statement

The GET statement is described in Chapter 12, "Stream-oriented data transmission" on page 252.

GO TO statement

The GO TO statement transfers control to the statement identified by the specified label reference. The GO TO statement is an unconditional branch.

The syntax for the GO TO statement is:



Abbreviation: GOTO

label specifies a label constant, a label variable, or a function reference that returns a label value. Since a label variable can have different values at each execution of the GO TO statement, control might not always transfer to the same statement.

If a GO TO statement transfers control from within a block to a point not contained within that block, the block is terminated. If the transfer point is contained in a block that did not directly activate the block being terminated, all intervening blocks in the activation sequence are also terminated (see "Procedure termination" on page 99).

When a GO TO statement specifies a label constant contained in a block that has more than one activation, control is transferred to the activation current when the GO TO is executed (see "Recursive procedures" on page 100).

A GO TO statement cannot transfer control:

- To an inactive block. Detection of such an error is not guaranteed.
- From outside a do-group to a statement inside a Type 2 or Type 3 do-group, unless the GO TO terminates a procedure or ON-unit invoked from within the do-group.
- To a FORMAT statement.

If the destination of the GO TO is specified by a label variable, it can then be used as a switch by assigning label constants to the label variable. If the label variable is subscripted, the switch can be controlled by varying the subscript. By using label variables or function references, quite complex switching can be effected. It is usually true, however, that simple control statements are the most efficient. GOTO operations from one block to another block hinder many optimizations in the target block, unless the target label is the last statement in its block.

%GO TO statement

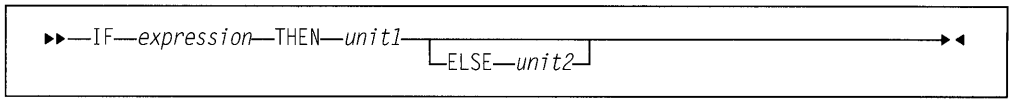
The %GO TO macro facility statement is discussed in "%GO TO" on page 456.

IF statement

The IF statement evaluates an expression and controls the flow of execution according to the result of that evaluation. The IF statement thus provides a conditional branch.

Note: Condition prefixes are invalid on IF and ELSE statements.

The syntax for the IF statement is:



expression
is evaluated and, if necessary, converted to a bit string.

When expressions involve the use of the & and/or | operators, they can be evaluated in any order. For more information on evaluations of expression, see Chapter 4, "Expressions and references" on page 50.

unit Each unit is either a valid single statement, a group, or a begin block. All single statements are considered valid and executable except DECLARE, DEFAULT, END, FORMAT, PROCEDURE, or a % statement. If a nonexecutable statement is used, the result can be unpredictable. Each unit can contain statements that specify a transfer of control (for example, GO TO). Hence, the normal sequence of the IF statement can be overridden.

Each unit can be labeled and can have condition prefixes.

IF is a compound statement. The semicolon terminating the last unit also terminates the IF statement.

If any bit in the string expression has the value '1'B, *unit1* is executed and *unit2*, if present, is ignored. If all bits are '0'B, or the string is null, *unit1* is ignored and *unit2*, if present, is executed.

IF statements can be nested. That is, either unit can itself be an IF statement, or both can be. Since each ELSE is always associated with the innermost unmatched IF in the same block or do-group, an ELSE with a null statement might be required to specify a desired sequence of control. For example, if B and C are constants, the following example:

```
if A = B then
:
else
  if A = C then
  :
  else
  :
```

is equivalent to and would be better coded as:

```
select( A );
  when ( B )
  :
  when ( C )
  :
  :
end;
```

Examples

In the following example, if the comparison is true (if A is equal to B), the value of D is assigned to C, and the ELSE unit is not executed.

```
if A = B then
  C = D;
else
  C = E;
```

If the comparison is false (A is not equal to B), the THEN unit is not executed, and the value of E is assigned to C.

Either the THEN unit or the ELSE unit can contain a statement that transfers control, either conditionally or unconditionally. If the THEN unit ends with a GO TO statement there is no need to specify an ELSE unit. For example:

```
if all(Array1 = Array2) then
  go to LABEL_1;
next-statement
```

If the expression is true, the GO TO statement of the THEN unit transfers control to LABEL_1. If the expression is not true, the THEN unit is not executed and control passes to the next statement.

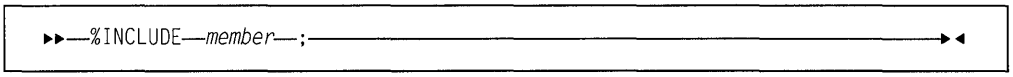
%IF statement

The %IF macro facility statement is discussed in "%IF" on page 456.

%INCLUDE statement

The %INCLUDE statement is used to incorporate external text into the source program.

The syntax for the %INCLUDE statement is:



member
specifies the first part of an OS/2 file name.

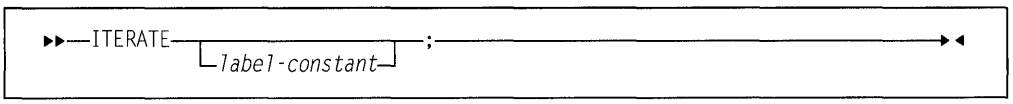
The included member can also contain %INCLUDE statements.

For information on using the %INCLUDE statement in applications, refer to *PL/I Package/2 Programming Guide*

ITERATE statement

The ITERATE statement transfers control to the END statement that delimits its containing iterative do-group. The current iteration completes and the next iteration, if needed, is started. It is valid only within an iterative do-group.

The syntax of the ITERATE statement is:



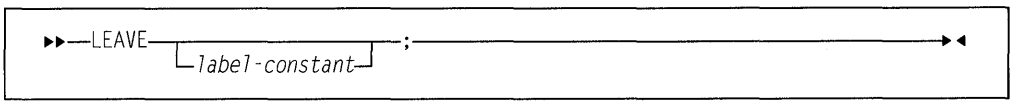
label-constant
Must be the label of a containing do-group. If omitted, control transfers to the END statement of the most recent iterative do-group containing the ITERATE statement.

For an example, see "Type 4" on page 179.

LEAVE statement

The LEAVE statement transfers control from within a do-group to the statement following the END statement that delimits the group and terminates the do-group. LEAVE is valid only within a do-group.

The syntax for the LEAVE statement is:



label-constant
must be a label of a containing do-group. The do-group that is left is the do-group that has the specified label. If *label-constant* is omitted, the do-group that is left is the group that contains the LEAVE statement.

The LEAVE statement and the referenced or implied DO statement must not be in different blocks.

In addition to the following examples, see the example in "Type 4" on page 179.

Example

In the following example, the statement `leave A;` transfers control to statement after group `A:`

```
A:  do I = 1 to 10;
       do J = 1 to 5;
          if X(I,J)=0 then
             leave A;
          else  /* ... */ ;
       end;
       statement within group A;
    end;
    statement after group A;
```

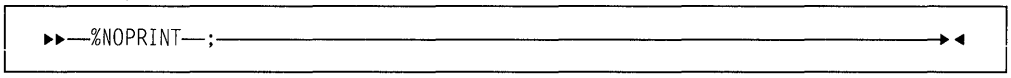
LOCATE statement

The LOCATE statement is described in Chapter 11, "Record-oriented data transmission" on page 242.

%NOPRINT statement

The %NOPRINT statement causes printing of the source listings to be suspended until a %PRINT statement is encountered or until a %POP statement is encountered that restores the previous %PRINT statement.

The syntax for the %NOPRINT statement is:

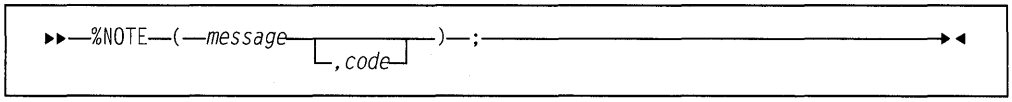


For an example of the %NOPRINT statement, refer to "%PUSH statement" on page 191.

%NOTE statement

The %NOTE statement generates a diagnostic message of specified text and severity.

The syntax for the %NOTE statement is:



message
a character expression whose value is the required diagnostic message.

code a fixed expression whose value indicates the severity of the message, as follows:

Code	Severity
0	I
4	W
8	E
12	S
16	U

If code is omitted, the default is 0.

If code has a value other than those listed above, a diagnostic message is produced and a default value is taken. If the value is less than 0 or greater than 16, severity U is the default. Otherwise, the next lower severity is the default.

Generated messages are filed together with other preprocessor messages. Whether or not a particular message is subsequently printed depends upon its severity level and the setting of the compiler FLAG option (as described in the *PL/I Package/2 Programming Guide*).

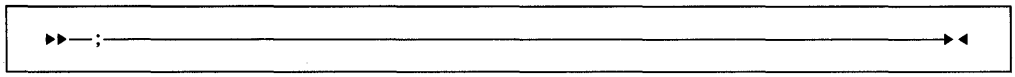
Generated messages of severity U cause immediate termination of preprocessing and compilation. Generated messages of severity S, E, or W might cause termination of compilation, depending upon the setting of various compiler options.

DBCS messages can be generated by using mixed data when the GRAPHIC compiler option is in effect.

null statement

The null statement causes no operation to be performed and does not modify sequential statement execution. It is often used to denote null action for THEN and ELSE clauses and for WHEN and OTHERWISE statements.

The syntax for the null statement is:



%null statement

The %null macro facility statement is discussed in "%null" on page 458.

ON statement

The ON statement is described in Chapter 15, "Condition handling" on page 302.

OPEN statement

The OPEN statement is described in Chapter 10, "Input and output" on page 228.

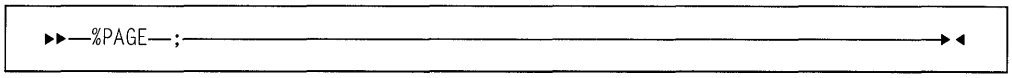
PACKAGE statement

The PACKAGE statement is described in "Packages" on page 92.

%PAGE statement

The %PAGE statement allows you to start a new page in the compiler source listings.

The syntax for the %PAGE statement is:

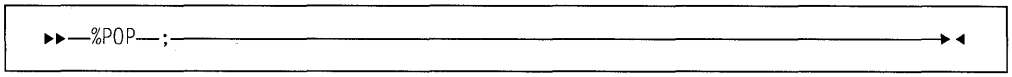


%POP statement

The %POP statement allows you to restore the status of the %PRINT and %NOPRINT statements saved by the most recent %PUSH statement.

The most common use of the %PUSH and %POP statements is in included files and macros.

The syntax of the %POP statement is:

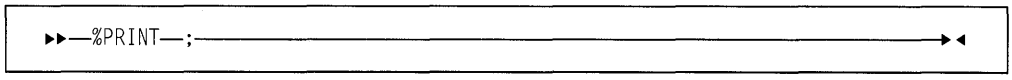


For an example, see "%PUSH statement" on page 191.

%PRINT statement

The %PRINT statement causes printing of the source listings to be resumed.

The syntax for the %PRINT statement is:



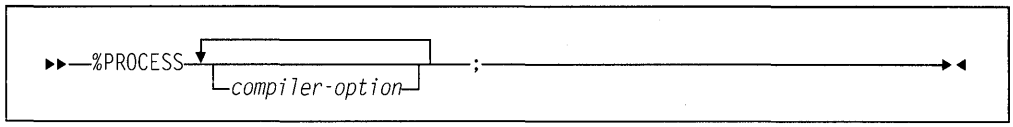
%PRINT is in effect, provided that the relevant compiler options are specified. For an example of the %PRINT statement, refer to "%PUSH statement" on page 191.

PROCEDURE statement

The PROCEDURE statement is described in "PROCEDURE statement" on page 94.

%PROCESS statement

The %PROCESS statement is used to override compiler options.



The % or * must be the first data byte of a source record. Any number of %PROCESS and *PROCESS statements can be specified, but they must all appear before the first language element appears. Refer to the *PL/I Package/2 Programming Guide* for more information.

*PROCESS statement

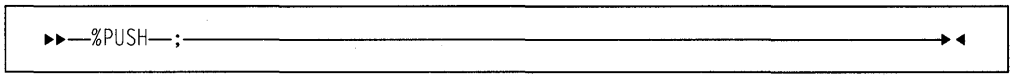
The *PROCESS statement is a synonym for the %PROCESS statement. For information on the %PROCESS statement, refer to on page 191.

%PUSH statement

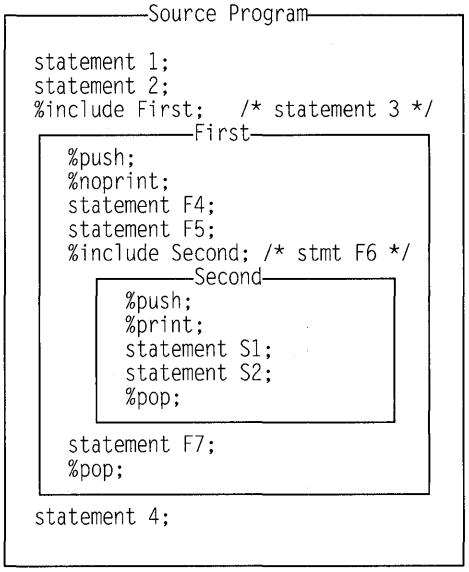
The %PUSH statement allows you to save the current status of the %PRINT and %NOPRINT statements in a "push down" stack on a last-in, first-out basis. You can restore this saved status later, also on a last-in, first-out basis, by using the %POP statement.

A common use of %PUSH and %POP statements is in included files and macros.

The syntax of the %PUSH statement is:



In the following example, statements 1, 2, 3, S1, S2, and 4 are printed in the listings. All others are not printed.



PUT statement

The PUT Statement is described in Chapter 12, "Stream-oriented data transmission" on page 252.

READ statement

The READ statement is described in Chapter 11, "Record-oriented data transmission" on page 242.

RELEASE statement

The RELEASE statement is described in "RELEASE statement" on page 103.

RESIGNAL statement

The RESIGNAL statement is described in "RESIGNAL statement" on page 309.

RETURN statement

The RETURN statement is described in "RETURN statement" on page 121.

REVERT statement

The REVERT statement is described in Chapter 15, "Condition handling" on page 302.

REWRITE statement

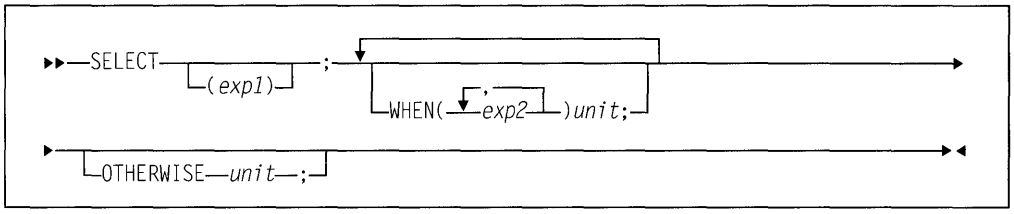
The REWRITE statement is described in Chapter 11, "Record-oriented data transmission" on page 242.

SELECT statement

A select-group provides a multiple path conditional branch. A select-group contains a SELECT statement, optionally one or more WHEN statements, optionally an OTHERWISE statement, and an END statement.

Note: Condition prefixes are invalid on SELECT, OTHERWISE, or WHEN statements.

The syntax for the select-group is:



Abbreviation: OTHER for OTHERWISE

SELECT (exp1)
and its corresponding END statement, delimit a group of statements collectively called a select-group. The expression in the SELECT statement is evaluated and its value is saved.

WHEN (exp2, exp2...) unit
specifies an expression or expressions that are evaluated and compared with the saved value from the SELECT statement. If an expression is found that is equal to the saved value, the evaluation of expressions in WHEN statements is terminated, and the unit of the associated WHEN statement is executed. If no such expression is found, the unit of the OTHERWISE statement is executed.

The WHEN statement must not have a label or condition prefix.

OTHERWISE unit
specifies the unit to be executed when every test of the preceding WHEN statements fails.

If the OTHERWISE statement is omitted and execution of the select-group does not result in the selection of a unit, the ERROR condition is raised.

The OTHERWISE statement must not have a label or condition prefix.

unit Each unit is either a valid single statement, a group, or a begin block. DECLARE, DEFAULT, END, FORMAT, PROCEDURE, and % statement statements are not valid. Each unit can contain statements that specify a transfer of control (for example, GO TO). Hence, the normal sequence of the SELECT statement can be overridden.

If *exp1* is omitted, each *exp2* is evaluated and converted, if necessary, to a bit string. If any bit in the resulting string is '1'B, the unit of the associated WHEN

statement is executed. If all bits are 0 or the string is null, the unit of the OTHERWISE statement is executed.

After execution of a unit of a WHEN or OTHERWISE statement, control passes to the statement following the select-group, unless the normal flow of control is altered within the unit.

If *exp1* is specified, each *exp2* must be such that the comparison expression

```
(exp1) = (exp2)
```

has a scalar bit value.

Array, structure, and union operands cannot be used in either *exp1* or *exp2*.

Examples

In the following example, `E`, `E1`, and so on, are expressions. When control reaches the SELECT statement, the expression `E` is evaluated and its value is saved. The expressions in the WHEN statements are then evaluated in turn (in the order in which they appear), and each value is compared with the value of `E`.

If a value is found that is equal to the value of `E`, the action following the corresponding THEN statement is performed; no further WHEN statement expressions are evaluated.

If none of the expressions in the WHEN statements are equal to the expression in the SELECT statement, the action specified after the OTHERWISE statement is executed.

```
select (e);
  when (e1,e2,e3) action-1;
  when (e4,e5) action-2;
  otherwise action-n;
end;
N1: next statement;
```

An example of *exp1* being omitted is:

```
select;
  when (A>B) call Bigger;
  when (A=B) call Same;
  otherwise call Smaller;
end;
```

If a select-group contains no WHEN statements, the action in the OTHERWISE statement is executed unconditionally. If the OTHERWISE statement is omitted, and execution of the select-group does not result in the selection of a WHEN statement, the ERROR condition is raised.

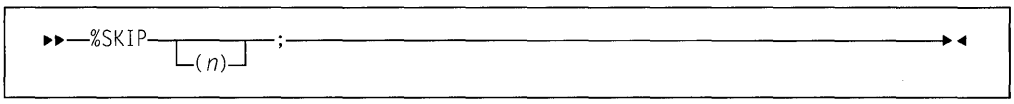
SIGNAL statement

The SIGNAL statement is described in Chapter 15, "Condition handling" on page 302.

%SKIP statement

The %SKIP statement causes the specified number of lines to be left blank in the compiler source listings.

The syntax for the %SKIP statement is:

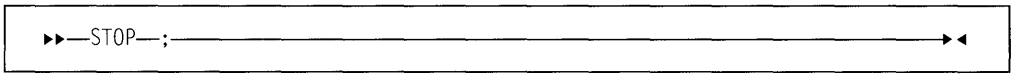


n specifies the number of lines to be skipped. It must be an integer in the range 1 through 999. If *n* is omitted, the default is 1. If *n* is greater than the number of lines remaining on the page, the equivalent of a %PAGE statement is executed in place of the %SKIP statement.

STOP statement

The STOP statement immediately terminates the program.

The syntax for the STOP statement is:



Prior to any termination activity, the FINISH condition is raised. On normal return from the FINISH ON-unit the program terminates.

WRITE statement

The WRITE statement is described in Chapter 11, "Record-oriented data transmission" on page 242.

Chapter 9. Storage Control

Chapter 9. Storage control

All variables require storage. The attributes specified for a variable describe the amount of storage required and how it is interpreted. In the following example a reference to X is a reference to a piece of storage that contains a value to be interpreted as fixed-point binary.

```
dcl X fixed binary(31,0) automatic;
```

Since X is automatic, the storage for it is allocated when its declaring block is activated, and the storage remains allocated until the block is deactivated.

Storage classes, allocation, and deallocation

Storage allocation is the process of associating an area of storage with a variable so that the data item(s) represented by the variable can be recorded internally. When storage is associated with a variable, the variable is *allocated*. Allocation for a given variable can take place *statically*, (before the execution of the program) or *dynamically* (during execution). A variable that is allocated statically remains allocated for the duration of the application program. A variable that is allocated dynamically relinquishes its storage either upon the termination of the block containing that variable, or at an explicit request from the application.

The storage class assigned to a variable determines the degree of storage control applied to it and the manner in which the variable's storage is allocated and freed. There are four storage classes: automatic, static, controlled, and based. You assign the storage class using its corresponding attribute in an explicit, implicit, or contextual declaration:

- AUTOMATIC specifies that storage is allocated upon each entry to the block that contains the storage declaration. The storage is released when the block is exited. If the block is a procedure that is invoked recursively, the previously allocated storage is pushed down upon entry; the latest allocation of storage is popped up in a recursive procedure when each generation terminates. (For a discussion of push-down and pop-up stacking, see "Recursive procedures" on page 100.)
- STATIC specifies that storage is allocated when the program is loaded. The storage is not freed until program execution is completed. The storage for a fetched procedure is not freed until the procedure is released.
- CONTROLLED specifies that you use the ALLOCATE and FREE statements to control the allocation and freeing of storage. Multiple allocations of the same controlled variable in the same program, without intervening freeing, will stack generations of the variable. You can access earlier generations only by freeing the later ones.
- BASED, like CONTROLLED, specifies that you control storage allocation and freeing. One difference is that multiple allocations are not stacked but are available at any time. Each allocation can be identified by the value of a pointer variable. Another difference is that based variables can be associated with an area of storage and identified by the value of an offset variable.

Storage class attributes can be declared explicitly for element, array, and major structure and union variables. For array and major structure and union variables,

the storage class declared for the variable applies to all of the elements in the array or structure or union.

Storage class attributes cannot be specified for:

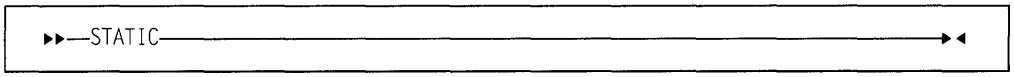
- Entry constants
- File constants
- Format constants
- Label constants
- CONDITION conditions
- Members of structures and unions
- Defined data items.

Allocation of storage for variables is managed by PL/I. You do not specify where in storage the allocation is to be made. You can, however, specify that a variable be allocated in an existing AREA. For more information, refer to "Area data and attribute" on page 211.

Static storage and attribute

Variables declared with the STATIC attribute are allocated prior to running a program. They remain allocated until the program terminates. The program has no control on the allocation of static variables during execution.

The syntax for the STATIC attribute is:



STATIC is the default for external variables, but internal variables can also be static. It is also the default for variables declared in a package, outside of any procedure. Static variables follow the normal scope rules for the validity of references to them. In the following example the variable X is allocated for the life of the program, but it can be referenced only within procedure B or any block contained in B. The variable Y gets the STATIC attribute and is also allocated for the life of the program.

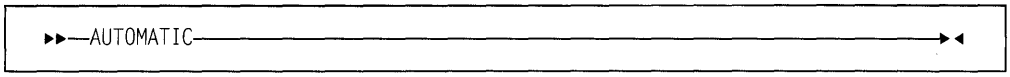
```
Package: Package exports (*);
  dcl Y char(10);

  A: proc options(main);
    B: proc;
      declare X static internal;
    end B;
  end A;

  C: proc;
    Y = 'hello';
  end C;

end Package;
```

If static variables are initialized using the INITIAL attribute, the initial values must be restricted expressions. Extent specifications must also be restricted expressions.

Automatic storage and attribute

Automatic variables are allocated on entry to the block in which they are declared. They can be reallocated many times during the execution of a program. You control their allocation by your design of the block structure.

The syntax for the AUTOMATIC attribute is:



Abbreviation: AUTO

AUTOMATIC is the default. Automatic variables are always internal.

In the following example, Each time procedure B is invoked, the variables X and Y are allocated storage. When B terminates, the storage is released, and the values X and Y contain are lost.

```
A: proc;
  ⋮
  call B;
  B: proc;
    declare X,Y auto;
    ⋮
    end B;
  ⋮
  call B;
```

The storage that is freed is available for allocation to other variables. Thus, whenever a block (procedure or begin) is active, storage is allocated for all variables declared automatic within that block. Whenever a block is inactive, no storage is allocated for the automatic variables in that block. Only one allocation of a particular automatic variable can exist, except for those procedures that are called recursively or by more than one program.

Extents for automatic variables can be specified as expressions. This means that you can allocate a specific amount of storage when you need it. In the following example the character string STR has a length defined by the value of the variable N when procedure B is invoked.

```
A: proc;
  declare N fixed bin;
  ⋮
  B: proc;
    declare STR char(N);
```

If the declare statements are located in the same block, PL/I requires that the variable N be initialized either to a restricted expression or to an initialized static variable. In the following example, the length allocated is correct for Str1, but not for Str2. PL/I does not resolve this type of declaration dependency.

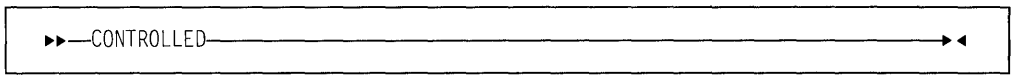
```
dcl N fixed bin (15) init(10),
    M fixed bin (15) init(N),
    Str1 char(N),
    Str2 char(M);
```

Controlled storage and attribute

Variables declared as CONTROLLED are allocated only when you specify them in an ALLOCATE statement. A controlled variable remains allocated until a FREE statement that names the variable is encountered or until the end of the program.

Controlled variables are partially independent of the program block structure, but not completely. The scope of a controlled variable can be internal or external. When it is declared as internal, the scope of the variable is the block in which the variable is declared and any contained blocks. Any reference to a controlled variable that is not allocated is in error.

The syntax for the CONTROLLED attribute is:



Abbreviation: CTL

In the following example the variable X can be validly referred to within procedure B and that part of procedure A that follows execution of the CALL statement.

```
A: proc;
  dcl X controlled;
  call B;
  ⋮
  B: proc;
    allocate X;
    ⋮
  end B;
end A;
```

Generally, controlled variables are useful when a program requires large data aggregates with adjustable extents. Statements in the following example allocate the exact storage required depending on the input data and free the storage when it is no longer required.

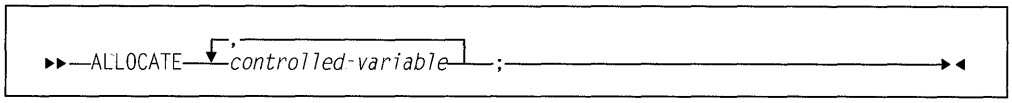
```
dcl A(M,N) ctl;
get list(M,N);
allocate A;
get list(A);
⋮
free A;
```

This method is more efficient than the alternative of setting up a begin block, because block activation and termination are not required.

ALLOCATE statement for controlled variables

The ALLOCATE statement allocates storage for controlled variables, independent of procedure block boundaries. Controlled parameters may also be allocated.

The syntax for the ALLOCATE statement for controlled variables is:



Abbreviation: ALLOC

controlled-variable
a controlled variable must be a level-1 unsubscripted variable.

Both controlled and based variables can be allocated in the same statement. For the syntax of based variables, refer to "ALLOCATE statement for based variables" on page 208.

Bounds for arrays, lengths of strings, and sizes of areas are evaluated at the execution of an ALLOCATE statement. A DECLARE statement must specify any necessary dimension, size, or length attributes for a variable. Any expression taken from a DECLARE statement is evaluated at the point of allocation using the conditions enabled at the ALLOCATE statement. However, names in the expression refer to variables whose scope includes the DECLARE or DEFAULT statement.

Initial values are assigned to a variable upon allocation, if the variable has an INITIAL attribute in the DECLARE statement. Expressions in the INITIAL attribute are evaluated at the point of allocation, using the conditions enabled at the ALLOCATE statement, although the names are interpreted in the environment of the declaration. If initialization involves reference to the variable being allocated, the reference is to the new generation of the variable. For more information on initialization, refer to "INITIAL attribute" on page 220.

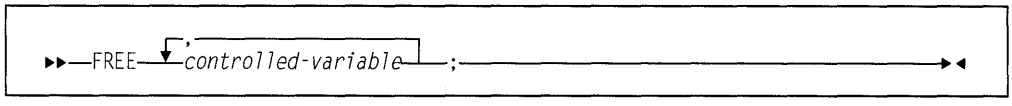
Any evaluations performed at the time the ALLOCATE statement is executed (for example, evaluation of expressions in an INITIAL attribute) must not be interdependent.

If storage for the controlled variable is not available, the STORAGE condition is raised.

FREE statement for controlled variables

The FREE statement frees the storage allocated for controlled variables. The freed storage is then available for other allocations. The previously allocated controlled variable is made available, and subsequent references refer to that allocation.

The syntax for the FREE statement for controlled variables is:



controlled-variable
is a level-1, unsubscripted variable.

Both based and controlled variables can be freed in the same statement. For the syntax of based variables, Refer to "FREE statement for based variables" on page 209.

Implicit freeing

A controlled variable need not be explicitly freed by a FREE statement. However, it is a good practice to explicitly FREE controlled variables.

All controlled storage is freed at the termination of the program.

Multiple generations of controlled variables

An ALLOCATE statement for a variable for which storage was previously allocated and not freed pushes down or stacks storage for the variable. This stacking creates a new generation of data for the variable. The new generation becomes the current generation. The previous generation cannot be directly accessed until the current generation has been freed. When storage for this variable is freed, using the FREE statement or at termination of the program in which the storage was allocated, storage is popped up or removed from the stack.

Adjustable extents

Controlled scalars, arrays, and members of structures and unions may have adjustable array extents, string lengths, and area sizes. In the following example, when the structure is allocated, `A.B` has the extent `-10` to `+10` and `A.C` is a VARYING character string with maximum length `5`.

```
dcl 1 A ctl,
      2 B(N:M),
      2 C char(L) varying;
N = -10;
M = 10;
L = 5;
alloc A;
free A;
```

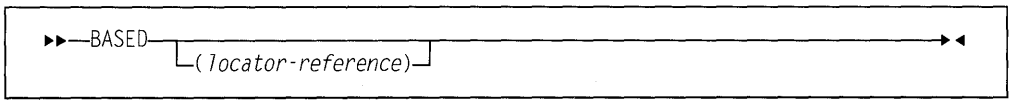
Built-in functions for controlled variables

The ALLOCATION built-in function can be used to determine the number of generations that have been allocated for a given controlled variable. If the variable is not allocated, the function returns the value zero.

Based storage and attribute

A declaration of a based variable is a description of the generation: the amount of storage required and its attributes. (A based variable does not identify the location of a generation in main storage.) A locator value identifies the location of the generation. Any reference to a based variable that is not allocated is in error.

The syntax for the BASED attribute is:



locator-reference
identifies the location of the data

When reference is made to a based variable, the data and alignment attributes used are those of the based variable, while the qualifying locator variable identifies the location of data.

A based variable cannot have the EXTERNAL attribute, but a locator reference for a based variable can have any storage class, including based.

A based structure or union can be declared to contain adjustable area sizes, array-bounds, and string-length specifications, by using the REFER option. See "REFER Option (Self-Defining Data)" on page 210.

A BASED VARYING string must have a maximum length equal to the maximum length of any string upon which it is defined.
For example:

```
declare A char(50) varying based(Q),
        B char(50) varying;
        Q=addr(B);
```

A based variable can be used to obtain storage by using the ALLOCATE statement or the LOCATE statement. A based variable can also be used to access existing data by using the READ statement (with SET option), or the FETCH statement (with SET option), or the ADDR built-in function.

Because a locator variable identifies the location of any generation, you can refer at any point in a program to any generation of a based variable by using an appropriate locator value. The following example declares that references to X, except when the reference is explicitly qualified, use the locator variable P to locate the storage for X.

```
dcl X fixed bin based(P);
```

The association of a locator reference in this way is not permanent. The locator reference can be used to identify locations of other based variables and other locator references can be used to identify other generations of the variable X. When a based variable is declared without a locator reference, any reference to the based variable must always be explicitly locator-qualified.

In the following example, the arrays A and C refer to the same storage. The elements B and C(2,1) also refer to the same storage.

```
dcl A(3,2) character(5) based(P),
    B char(5) based(Q),
    C(3,2) character(5);
P = addr(C);
Q = addr(A(2,1));
```

Note: When a based variable is overlaid in this way, no new storage is allocated. The based variable uses the same storage as the variable on which it is overlaid (`C(3,2)` in the example).

You can also use the DEFINED and UNION attributes to overlay variable storage, but DEFINED and UNION overlay the storage permanently. When based variables are overlaid with a locator reference, the association can be changed at any time in the program by assigning a new value to the locator variable.

For more information on the DEFINED and UNION attributes, refer to "DEFINED and POSITION attributes" on page 218, and "Unions" on page 149.

The INITIAL attribute can be specified for a based variable. The initial values are assigned only upon explicit allocation of the based variable with an ALLOCATE or LOCATE statement.

Locator data

There are two types of locator data: pointer and offset.

The value of a *pointer variable* is an address of a location in storage. It can be used to qualify a reference to a variable with allocated storage in several different locations.

The value of an *offset variable* specifies a location within an area variable and remains valid when the area is assigned to a different part of storage.

A locator value can be assigned only to a locator variable. When an offset value is assigned to an offset variable, the area variables named in the OFFSET attributes are ignored.

Locator conversion

Locator data cannot be converted to other data types, except as follows:

- To and from REAL FIXED BINARY (p,0) by using the BINARYVALUE, POINTERVALUE, and OFFSETVALUE built-in functions
- Between pointer and offset implicitly or explicitly using the POINTER and OFFSET built-in functions.

When an offset variable is used in a reference, it is implicitly converted to a pointer value by using the address of the area variable designated in the OFFSET attribute and the offset variable. Explicit conversion of an offset to a pointer value is accomplished using the POINTER built-in function. For example, the following statement assigns a pointer value to `P`, giving the location of a based variable, identified by offset `O` in area `B`.

```
dcl P pointer, O offset(A),B area;
P = pointer(O,B);
```

Because the area variable is different from that associated with the offset variable, you must ensure that the offset value is valid for the different area. It is valid, for example, if area A is assigned to area B prior to the invocation of the function.

The OFFSET built-in function, in contrast to the POINTER built-in function, returns an offset value derived from a given pointer and area. The given pointer value must identify the location of a based variable in the given area.

A pointer value is converted to offset by using the pointer value and the address of the area. This conversion is limited to pointer values that relate to addresses within the area named in the OFFSET attribute.

Except when assigning the NULL or the SYSNULL built-in function value, it is an error to attempt to convert from or to an offset variable that is not associated with an area.

There is no implicit locator conversion in multiple assignments.

Locator reference

A locator reference is either a locator variable that can be qualified or subscripted, or a function reference that returns a locator value.

A locator reference can be used in the following ways:

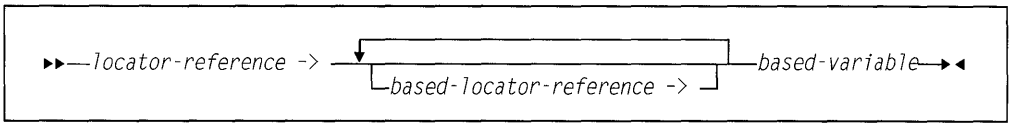
- As a locator qualifier, in association with a declaration of a based variable
- In a comparison operation, as in an IF statement
- As an argument in a procedure reference.

Because PL/I implicitly converts an offset to a pointer value, offset references can be used interchangeably with pointer references.

Locator qualification

Locator qualification is the association of one or more locator references with a based reference to identify a particular generation of a based variable. This is called a locator-qualified reference. The composite symbol –> represents "qualified by" or "points to."

The syntax for explicit qualified reference is:



locator-reference
based-locator-reference
identify the location of the data.

In the following example, X is a based variable, P is a locator variable, and Q is a based locator variable.

```
P -> Q -> X
```

The reference means that it is that generation of X that is identified by the based locator Q that is also identified by the value of the locator P. X and Q are said to be *explicitly locator-qualified.*

When more than one locator qualifier is used, they are evaluated from left to right.

Reference to a based variable can also be *implicitly qualified*. The locator reference used to determine the generation of a based variable that is implicitly qualified is the one declared with the based variable. In the following example, the ALLOCATE statement sets the pointer variable P so that the reference X applies to allocated storage.

```
dcl X fixed bin based(P) init(0);
allocate X;
X = X + 1;
```

The references to X in the assignment statement are implicitly locator-qualified by P. References to X can also be explicitly locator-qualified as shown in the following example.
The following assignment statements have the same effect as the previous example:

```
P->X = P->X + 1;
Q = P;
Q->X = Q->X + 1;
```

Because the locator declared with a based variable can also be based, a chain of locator qualifiers can be implied. For example, the following pointer and based variables can be used:

```
declare (P(10),Q) pointer,
         R pointer based (Q),
         V based (P(3)),
         W based (R),
         Y based;
allocate R,V,W;
```

Given the previous declaration and allocation, the following references are valid:

```
P(3) -> V
V
Q -> R -> W
R -> W
W
```

The first two references are equivalent, and the last three are equivalent. Any reference to Y must include a qualifying locator variable.

Levels of locator qualification

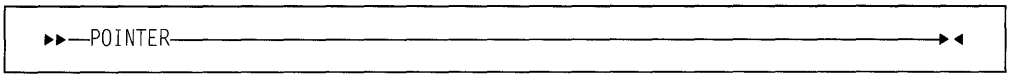
A pointer that qualifies a based variable represents one level of locator qualification. An offset represents two levels because it is implicitly qualified within an area. The number of levels is not affected by a locator being subscripted and/or an element of a structure or union. In the following example the references X, P -> X, and Q -> P -> X represent three levels of locator qualification.

```
declare X based (P),
        P pointer based (Q),
        Q offset (A);
```

POINTER variable and attribute

A pointer variable is declared contextually if it appears in the declaration of a based variable, as a locator qualifier, in a BASED attribute, or in the SET option of an ALLOCATE, LOCATE, READ, or FETCH statement. It can also be declared explicitly.

The syntax for the POINTER attribute is:



Abbreviation: PTR

The value of a pointer variable that no longer identifies a generation of a based variable is undefined (for example, when a based variable has been freed). Before a reference is made to a pointer-qualified variable, the pointer must have a value.

Built-in functions for based variables

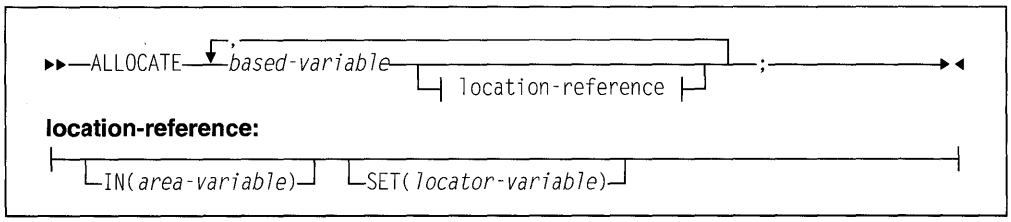
The ADDR built-in function returns a pointer value that identifies the first byte of a variable. The ENTRYADDR built-in function returns a pointer value that is the address of the first executed instruction if the entry were to be invoked.

Note: The NULL and SYSNULL built-in functions can, but do not necessarily, compare equally. Your application program must **not** depend on the functions' equality.

ALLOCATE statement for based variables

The ALLOCATE statement allocates storage for based variables and sets a locator variable that can be used to identify the location, independent of procedure block boundaries.

The syntax for the ALLOCATE statement for based variables is:



Abbreviation: ALLOC

based variable
is a level-1 unsubscripted variable.

IN specifies the area variable in which the storage is allocated. For more information on areas, refer to "Area data and attribute" on page 211.

SET specifies a locator variable that is set to the location of the storage allocated. If the SET option is not specified, the locator used will be that specified in the declaration of the based variable. For syntax information about declaring based variables, refer to "Based storage and attribute" on page 204 and "Locator data" on page 205.

Both based and controlled variables can be allocated in the same statement. For the syntax of controlled variables, see "ALLOCATE statement for controlled variables" on page 202.

Storage is allocated in an area when the IN option is specified or the SET option specifies an offset variable. These options can appear in any order. For allocations in areas:

- If sufficient storage for the based variable does not exist within the area, the AREA condition is raised.
- If the IN option is not used when using an offset variable, the declaration of the offset variable must specify an area reference.

When an area is not used, the locator variable must be a pointer variable. If storage for the based variable is not available, the STORAGE condition is raised.

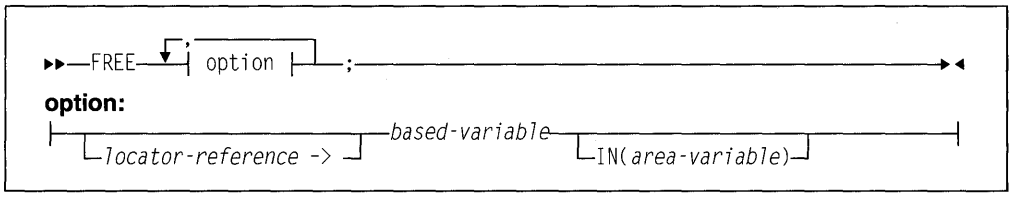
The amount of storage allocated for a based variable depends on its attributes, and on its dimensions, length, or size specifications if these are applicable at the time of allocation. These attributes are determined from the declaration of the based

A based structure or union can contain adjustable array bounds or string lengths or area sizes (see "REFER Option (Self-Defining Data)" on page 210). The asterisk notation for extents is not allowed for based variables.

FREE statement for based variables

The FREE statement frees the storage allocated for based and controlled variables.

The syntax for the FREE statement for based variables is:



locator-reference ->
frees a particular generation of a based variable. The composite symbol -> means "qualified by" or "points to." If the based variable is not explicitly locator-qualified, the locator variable declared in the BASED attribute is used to identify the generation of data to be freed. If no locator has been declared, the statement is in error.

based variable
must be a level-1 unsubscripted based variable.

IN must be specified or the based variable must be qualified by an offset declared with an associated area, if the storage to be freed was allocated in an area. The IN option cannot appear if the based variable was not allocated in an area. Area assignment allocates based storage in the target area. These allocations can be freed by the IN option naming the target area.

Both based and controlled variables can be freed in the same statement. For the syntax of controlled variables, see "FREE statement for controlled variables" on page 202.

A based variable can be used to free storage only if that storage has been allocated for a based variable having identical data attributes.

The amount of storage freed depends upon the attributes of the based variable, including bounds and/or lengths at the time the storage is freed. The user is responsible for determining that this amount coincides with the amount allocated. If the variable has not been allocated, the results are unpredictable.

Implicit freeing

A based variable need not be explicitly freed by a FREE statement, but it is a good practice to do so.

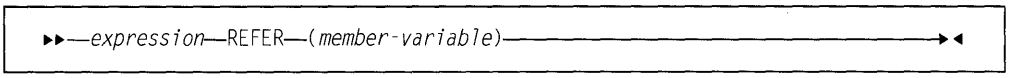
All based storage is freed at the termination of the program.

REFER Option (Self-Defining Data)

A self-defining structure or union contains information about its own fields, such as the length of a string. A based structure or union can be declared to manipulate this data. String lengths, array bounds, and area sizes can all be defined by variables, known as the *refer object*, declared within the structure or union. When the structure or union is allocated (by either an ALLOCATE statement or a LOCATE statement), the value of an expression is assigned to the refer object variable. For any other reference to the structure or union, the value of the refer object is used.

The REFER option is used in the declaration of a based structure or union to specify that, on allocation of the structure or union, the value of an expression is assigned to the refer object and represents the length, bound, or size of another variable in the structure or union.

The syntax for a length, bound, or size with a REFER option is:



expression

The value of this expression defines the length, bound, or size of the member when the structure or union is allocated (using ALLOCATE or LOCATE). The expression is evaluated and converted to FIXED BINARY (M,0). Any variables used as operands in the expression must not belong to the structure or union containing the REFER option.

Subsequent references to the structure or union obtain the REFER option member's length, bound, or size from the current value of *member-variable* (refer object).

member-variable

The refer object must conform to the following rules:

- It must be a member of the same level-1 structure or union.
- It must be REAL FIXED BINARY (p,0).

- It cannot be locator-qualified (see "Locator qualification" on page 206) or subscripted.
- It cannot be part of an array.
- Explicit assignment to it is not allowed. That is, the variable is given the NONASSIGNABLE attribute.
- It must precede all members that have the REFER option.

In the following example, the declaration specifies that the based structure STR consists of an array Y and an element X.

```
declare 1 STR based(P),
          2 X fixed binary(31,0),
          2 Y (L refer (X)),
        L fixed binary(31,0) init(1000);
```

When STR is allocated, the upper bound is set to the current value of L which is assigned to X. For any other reference to Y, such as a READ statement that sets P, the bound value is taken from X.

If the INITIAL attribute is specified for the member with the REFER option, initialization of the member occurs after the refer object has been assigned its value.

Any number of REFER options can be used in the declaration of a structure or union.

The value of the refer object should not be changed during program execution. It is an error to free such an aggregate if the value of the refer object has changed.

Area data and attribute

Area variables describe areas of storage that are reserved for the allocation of based variables. This reserved storage can be allocated to, and freed from, based variables by the ALLOCATE and FREE statements. Area variables can have any storage class and must be aligned.

When a based variable is allocated and an area is not specified, the storage is obtained from wherever it is available. Consequently, allocated based variables can be scattered widely throughout main storage. This is not significant for internal operations because items are readily accessed using the pointers. However, if these allocations are transmitted to a data set, the items have to be collected together. Items allocated within an area variable are already collected and can be transmitted or assigned as a unit while still retaining their separate identities.

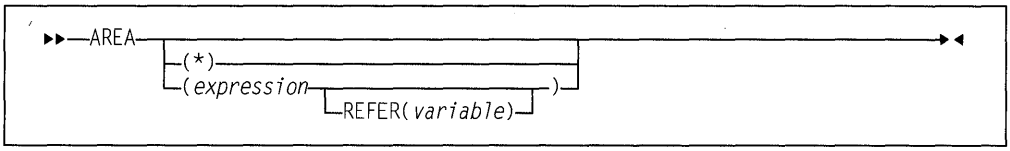
You might want to identify the locations of based variables within an area variable relative to the start of the area variable. Offset variables are provided for this purpose.

An area can be assigned or transmitted complete with its contained allocations; thus, a set of based allocations can be treated as one unit for assignment and input/output while each allocation retains its individual identity.

The size of an area is adjustable in the same way as a string length or an array bound and therefore it can be specified by an expression or an asterisk (for a controlled area parameter) or by a REFER option (for a based area).

A variable is given the AREA attribute contextually by its appearance in the OFFSET attribute or an IN option, or by explicit declaration.

The syntax for the AREA attribute is:



expression
specifies the size of the area. If *expression*, or an asterisk is not specified, the default is 1000.

***** An asterisk can be used to specify the size if the area variable is declared is a parameter.

REFER For a description of the REFER option, refer to "REFER Option (Self-Defining Data)" on page 210.

The area size for areas that have the storage classes AUTOMATIC or CONTROLLED is given by an expression whose value specifies the number of reserved bytes.

If an area has the BASED attribute, the area size must be a constant unless the area is a member of a based structure or union and the REFER option is used.

The size for areas of static storage class must be specified as a restricted expression.

Examples of AREA declarations are:

```
declare area1 area(2000),
        area2 area;
```

In addition to the declared size, an extra 16 bytes of control information precedes the reserved size of an area. The 16 bytes contain such details as the amount of storage in use.

The amount of reserved storage that is actually in use is known as the *extent* of the area. When an area variable is allocated, it is empty, that is, the area extent is zero. The maximum extent is represented by the area size. Based variables can be allocated and freed within an area at any time during execution, thus varying the extent of an area.

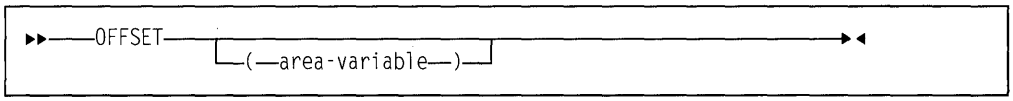
When a based variable is freed, the storage it occupied is available for other allocations. A chain of available storage within an area is maintained; the head of the chain is held within the control information. Inevitably, as based variables with different storage requirements are allocated and freed, gaps will occur in the area when allocations do not fit available spaces. These gaps are included in the extent of the area.

No operators, including comparison, can be applied to area variables.

Offset data and attribute

Offset data is used exclusively with area variables. The value of an offset variable indicates the location of a based variable within an area variable relative to the start of the area. Because the based variables are located relatively, if the area variable is assigned to a different part of main storage, the offset values remain valid.

Offset variables do not preclude the use of pointer variables within an area. The syntax for the OFFSET attribute is:



The association of an area variable with an offset variable is not permanent. An offset variable can be associated with any area variable by means of the POINTER built-in function (see "Locator conversion" on page 205). The advantage of making such an association in a declaration is that a reference to the offset variable implies reference to the associated area variable. If no area variable is specified, the offset can be used as a locator qualifier only through use of the POINTER built-in function.

Setting offset variables

The value of an offset variable can be set in any one of the following ways:

- By an ALLOCATE statement
- By assignment of the value of another locator variable, or a locator value returned by a user-defined function
- The NULL, SYSNULL, ADDR, ENTRYADDR, OFFSETADD, OFFSETSUBTRACT, OFFSETVALUE, or OFFSET built-in function

If no area variable is specified, the offset can be used only as a locator qualifier through use of the POINTER built-in function.

Examples of offset variables

Consider the following example:

```
dcl X based(O),
    Y based(P),
    A area,
    O offset(A);

    allocate X;
    allocate Y in(A);
```

The storage class of area A and offset O is AUTOMATIC by default. The first ALLOCATE statement is equivalent to:

```
ALLOCATE X IN(A) SET(O);
```

The second ALLOCATE statement is equivalent to:

```
ALLOCATE Y IN(A) SET(P);
```

The following example shows how a list can be built in an area variable using offset variables:

```
dcl A area,
    (T,H) offset(A),
    1 STR based(H),
      2 P offset(A),
      2 data;

    allocate STR in(A);
    T=H;

  do loop;
    allocate STR set(T->P);
    T=T->P;
    .
    .
  end;
```

Area assignment

The value of an area reference can be assigned to one or more area variables by an assignment statement. Area-to-area assignment has the effect of freeing all allocations in the target area and then assigning the extent of the source area to the target area, so that all offsets for the source area are valid for the target area. In the following example:

```
declare X based (O(1)),
        O(2) offset (A),
        (A,B) area;

        alloc X in (A);
        X = 1;
        alloc X in (A) set (O(2));
        O(2) -> X = 2;
        B = A;
```

Using the POINTER built-in function, the references `POINTER (O(2),B)->X` and `O(2)->X` represent the same value allocated in areas B and A respectively.

If an area containing no allocations is assigned to a target area, the effect is to free all allocations in the target area.

Area assignment can be used to expand a list of based variables beyond the bounds of its original area. If you attempt to allocate a based variable within an area that contains insufficient free storage to accommodate it, the AREA condition is raised. The ON-unit for this condition can be to change the value of a pointer qualifying the reference to the inadequate area, so that it points to a different area; on return from the ON-unit, the allocation is attempted again, within the new area. Alternatively, the ON-unit can write out the area and reset it to EMPTY.

Input/output of areas

Areas allow input and output of complete lists of based variables as one unit, to and from RECORD files. On output, the area extent, together with the 16 bytes of control information, is transmitted, except when the area is in a structure or union and is not the last item in it—then, the declared size is transmitted. Thus the unused part of an area does not take up space on the data set.

Because the extents of areas can vary, varying length records should be used. The maximum record length required is governed by the area length (area size + 16).

List processing

List processing is the name for a number of techniques to help manipulate collections of data. Although arrays, structures, and unions are also used for manipulating collections of data, list processing techniques are more flexible since they allow collections of data to be indefinitely reordered and extended during program execution. The purpose here is not to illustrate these techniques but is to show how based variables and locator variables serve as a basis for this type of processing.

In list processing, a number of based variables with many generations can be included in a list. Members of the list are linked together by one or more pointers in one member identifying the location of other members or lists. The allocation of a based variable cannot specify where in main storage the variable is to be allocated (except that you can specify the area that you want it allocated in). In practice, a chain of items can be scattered throughout main storage, but by accessing each pointer the next member is found. A member of a list is usually a structure or union that includes a pointer variable.
The following example creates a list of structures:

```
dcl 1 STR based(H),
      2 P pointer,
      2 data,
    T  pointer;

      allocate STR;
      T=H;

      do loop;
        allocate STR set(T->P);
          T=T->P;
          T->P=null;
          .
          .
          .
       end;
```

The structures are generations of STR and are linked by the pointer variable P in each generation. The pointer variable T identifies the previous generation during the creation of the list. The first ALLOCATE statement sets the pointer H to identify it. The pointer H identifies the start, or head, of the list. The second ALLOCATE statement sets the pointer P in the previous generation to identify the location of this new generation. The assignment statement T=T->P; updates pointer T to identify the location of the new generation. The assignment statement T->P=NULL; sets the pointer in the last generation to NULL, giving a positive indication of the end of the list.

Figure 38 on page 216 shows a diagrammatic representation of a one-directional chain.

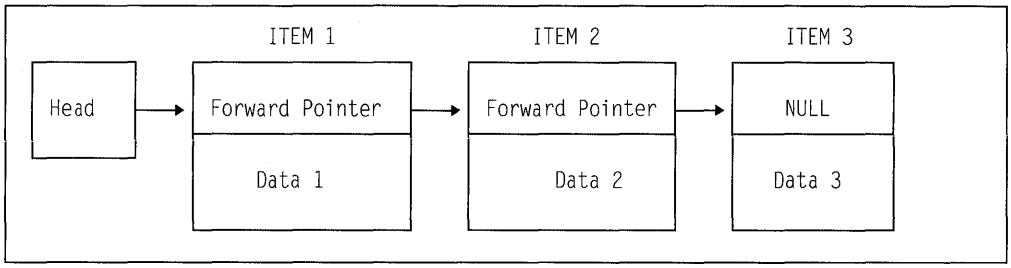


Figure 38. Example of One-Directional Chain

Unless the value of P in each generation is assigned to a separate pointer variable for each generation, the generations of STR can be accessed only in the order in which the list was created. For the above example, the following statements can be used to access each generation in turn:

```
do T=H
  repeat(T->P)
  while (T¬=null);
  ...
  ... T->data ...;
  ...
end;
```

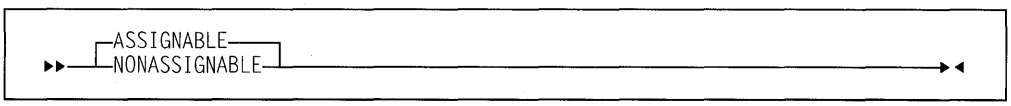
The foregoing examples show a simple list processing technique, the creation of a unidirectional list. More complex lists can be formed by adding other pointer variables into the structure or union. If a second pointer is added, it can be made to point to the previous generation. The list is then bidirectional; from any item in the list, the previous and next items can be accessed by using the appropriate pointer value. Instead of setting the last pointer value to the value of NULL, it can be set to point to the first item in the list, creating a ring or circular list.

A list need not consist only of generations of a single based variable. Generations of different based structure or unions can be included in a list by setting the appropriate pointer values. Items can be added and deleted from a list by manipulating the values of pointers. A list can be restructured by manipulating the pointers so that the processing of data in the list can be simplified.

ASSIGNABLE and NONASSIGNABLE attributes

The ASSIGNABLE and NONASSIGNABLE attributes specify whether the associated variable can be the target of an assignment.

The syntax for the ASSIGNABLE and NONASSIGNABLE attributes is:



Abbreviations: ASGN, NONASGN

Default: ASSIGNABLE

If a variable has the NONASSIGNABLE attribute, the variable cannot be assigned to.

If an entry descriptor has the NONASSIGNABLE attribute, the argument is assumed not to change when the associated ENTRY is invoked. If the argument is a constant, no dummy argument is created.

The ASSIGNABLE and NONASSIGNABLE attributes are propagated to members of structures or unions.

NORMAL and ABNORMAL attributes

The NORMAL and ABNORMAL attributes specify whether the associated variable is subject to change any time.

The ABNORMAL attribute specifies that the value of the variable can change between statements or within a statement. An abnormal variable is fetched from or stored in storage each time it is needed or each time it is changed. All optimization is inhibited for an abnormal variable.

The syntax for the NORMAL and ABNORMAL attributes is:

Default: NORMAL

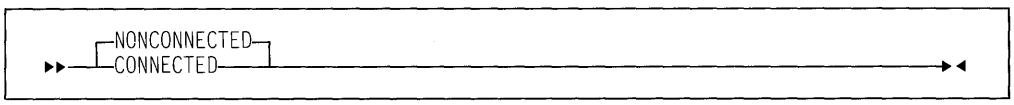
The NORMAL and ABNORMAL attributes are propagated to members of structures or unions.

CONNECTED and NONCONNECTED attributes

Elements, arrays, and major structure or unions are always allocated in connected storage. References to unconnected storage arise only when you refer to an aggregate that is made up of noncontiguous items from a larger aggregate. (See "Cross sections of arrays" on page 147.) For example, in the following structure the interleaved arrays `A.B` and `A.C` are both in unconnected storage.

```
1 A(10),
  2 B,
  2 C;
```

The syntax for the CONNECTED and UNCONNECTED attributes is:



Abbreviations: CONN, NONCONN

Default: NONCONNECTED

The CONNECTED attribute is applicable only to noncontrolled aggregate parameters and can be specified only on level-1 names. It specifies that the parameter is a reference to connected storage only, and therefore, allows the parameter to be used as a target or source in record-oriented I/O, or as a base in string overlay

defining. When the parameter is connected and the CONNECTED attribute is used, more efficient object code is produced for references to the connected parameter.

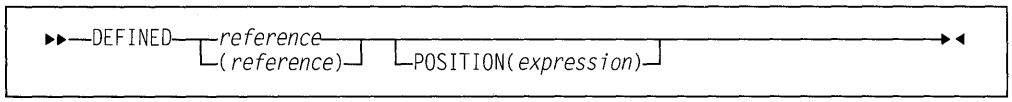
NONCONNECTED must be specified or defaulted if a parameter occupies noncontiguous storage. In the following example the NONCONNECTED attribute specifies that the sum_Slice routine handles 1-dimensional arrays in which the elements may not be contiguous. In the first invocation, sum_Slice is passed the first row, which is in connected storage. In the second invocation, however, sum_Slice is passed the first column, which is in nonconnected storage.

```
dcl A(10,10) fixed bin(31);

display( sum_Slice( A(1,*) ) );     /* first row    */
display( sum_Slice( A(*,1) ) );     /* first column */

sum_Slice:proc(X) returns(fixed bin(31));

  dcl X (*) fixed bin(31) nonconnected; /* default */
  return(sum(X) );
end;
```

DEFINED and POSITION attributes

The DEFINED attribute specifies that the declared variable is associated with some or all of the storage associated with the designated base variable. The UNION attribute allows you to achieve this and also permits variables with different attributes and precisions to be overlaid. Another difference is that DEFINED guarantees that access through defined or base variables is reflected in all defined variables. In a union only one member of the union is valid at any given time. For syntax information on the UNION attribute, refer to "UNION attribute" on page 149.

The POSITION attribute, which can be used only with string-overlay defining, specifies the bit, character, or graphic within the base variable at which the defined variable is to begin. The syntax for the DEFINED attribute is:



Abbreviation: DEF for DEFINED, POS for POSITION

reference

refers to the "base" variable, for which the storage is associated with the declared (defined) variable. The base variable can be EXTERNAL or INTERNAL. It can be a connected parameter. It cannot be BASED, DEFINED or CONTROLLED. A change to the base variable's value is a corresponding change to the value of the defined variable, and vice versa. The base variable can have adjustable extents. It should be specified as NONVARYING, with a data type of CHARACTER, BIT, or GRAPHIC. Other data types produce unpredictable results.

expression

specifies the position relative to the start of the base variable. If the POSITION attribute is omitted, POSITION(1) is the default. The value specified in the expression can range from 1 to *N*. The value *N* is defined as

`N = N(B) - N(D) + 1` where `N(B)` is the number of bits, characters, or graphics in the base variable, and `N(D)` is the number of bits, characters, or graphics in the defined variable.

The expression is evaluated and converted to an integer value at each reference to the defined item.

When the defined variable is a bit aggregate:

- The POSITION attribute can contain only a restricted expression.
- The base variable must not be subscripted.

The defined variable does not inherit any attributes from the base variable. It can have the dimension attribute. The defined variable:

- Must be INTERNAL and a level-1 identifier
- Cannot have INITIAL, AUTOMATIC, BASED, CONTROLLED, STATIC, PARAMETER, or ALIGNED attributes.

In references to defined data, the STRINGRANGE, SUBSCRIPTRANGE, and STRINGSIZE conditions are raised for the array bounds and string lengths of the defined variable, not the base variable.

Values are determined and names are interpreted as follows:

- The array bounds, string lengths, and area sizes of a defined variable must be known at compile time.
- A reference to a defined variable is a reference to the current generation of the base variable. When a defined variable is passed as an argument without the creation of a dummy argument, the corresponding parameter refers to the base variable when the argument is passed.

Both the defined and the base variables must belong to:

- The bit class, including:

  NONVARYING bit variables
  Aggregates of NONVARYING bit variables.

- The character class, including:

  NONVARYING character variables
  Character pictured and numeric pictured variables
  Aggregates of the previous two.

- The graphic class, including:

  NONVARYING graphic variables
  Aggregates of NONVARYING graphic variables.

Examples:

```
dcl A char(100),
    V(10,10) char(1) def A;
```

`V` is a two-dimensional array that consists of all elements in the character string `A`.

```
dcl B(10) char(1),
    W char(10) def B;
```

`W` is a character string that consists of all the elements in the array `B`.

```
dcl C char(10),
    X char(4) pos(7) def(C);
```

X is a character string representing the 4 rightmost characters of C.

```
dcl D(10,10) char(10),
    Y(5,5) char(2) pos(1) def(D),
    Y2(5,2,2) char(2) def(D);
```

Y and D match, so the POSITION attribute is required. However, POSITION is not required for Y2 because Y2 and D do not match.

```
dcl 1 E,
      2 (F,G,H) char(16),
    1 Z def(E) pos(1),
      2 (F,G,H) char(1),
    1 Z2 def(E),
      2 (F,G,H) char(1),
      2 (I,J) char(1);
```

Z and E match; therefore, POSITION is required. However, Z2 and E do not match.

```
dcl C(10,10) bit(1),
    X bit(40) def(C) pos(20);
```

X is a bit string that consists of 40 elements of C, starting at the twentieth element.

```
dcl E pic'99V.999',
    Z1(6) char(1) def(E),
    Z2 char(3) def(E) pos(4),
    Z3(4) char(1) def(E) pos(2);
```

Z1 is a character string array that consists of all the elements of the numeric picture E. Z2 is a character string that consists of the elements '999' of the picture E. Z3 is a character-string array that consists of the elements '9.99' of the picture E.

```
dcl A(20) char(10),
    B(10) char(5) def A position(1);
```

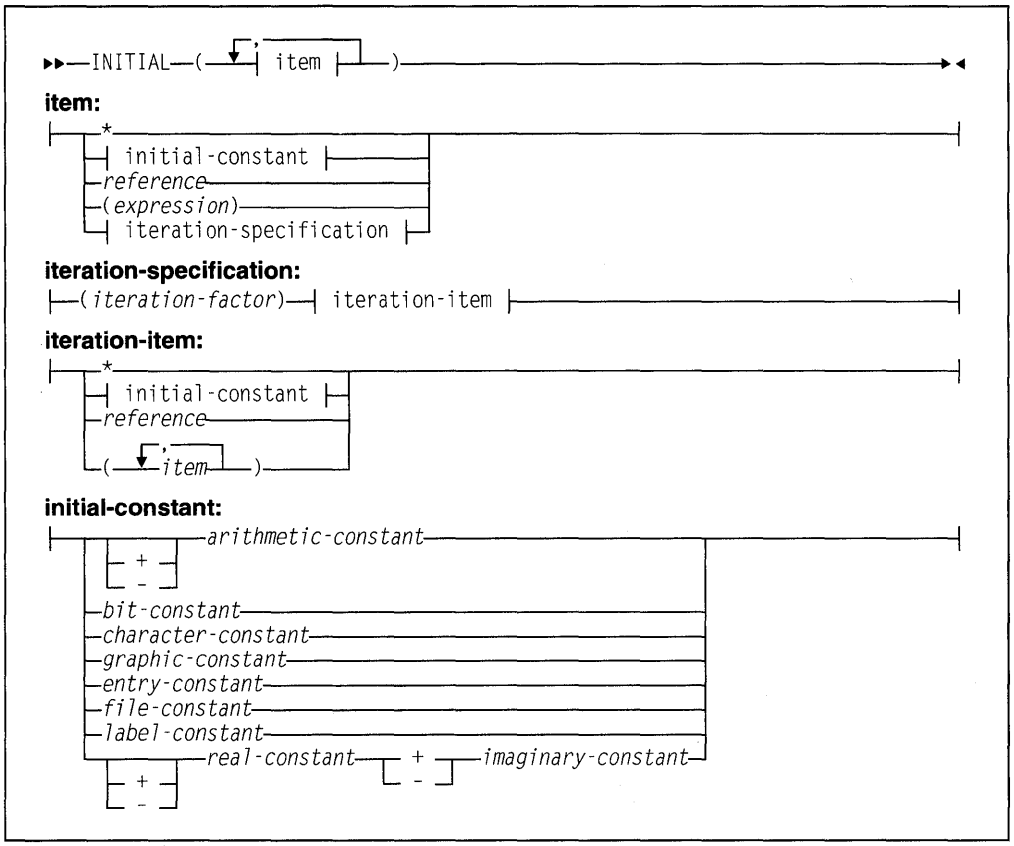
The first 50 characters of B consist of the first 50 characters of A. POSITION(1) must be explicitly specified because A and B match.

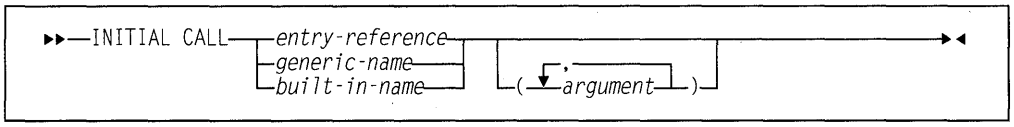
INITIAL attribute

The INITIAL attribute specifies an initial value or values assigned to a variable at the time storage is allocated for it. Only one initial value can be specified for an element variable. More than one can be specified for an array variable. A structure or union variable can be initialized only by separate initialization of its elementary names, whether they are element or array variables. The INITIAL attribute cannot be given to constants, defined data, noncontrolled parameters, and non-LIMITED static entry variables.

The INITIAL attribute has two forms. The first form, INITIAL, specifies an initial constant, expression, or function reference, for which the value is assigned to a variable when storage is allocated to it. The second form, INITIAL CALL, specifies (with the CALL option) that a procedure is invoked to perform initialization. The variable is initialized by assignment during the execution of the called routine. (The routine is not invoked as a function that returns a value to the point of invocation.)

The syntax for the INITIAL attribute is:





Abbreviations: INIT and INIT CALL

***** specifies that the element is to be left uninitialized.

iteration factor
specifies the number of times the iteration item is to be repeated in the initialization of elements of an array.

The iteration factor can be an expression or an asterisk.

- An expression is converted to FIXED BIN(31). For static variables, it must be a constant.
- An asterisk indicates that the remaining elements should be initialized to the specified value.

A negative or zero iteration factor specifies no initialization.

constant
reference
expression
specify an initial value to be assigned to the initialized variable.

INITIAL CALL

For INITIAL CALL, the entry reference and argument list passed must satisfy the condition stated for block activation as discussed under "Block activation" on page 91.

INITIAL CALL cannot be used to initialize static data.

The following example initializes all of the elements of A to X'00' without the need for the INITIAL attribute on each element:

```
dcl  1  A automatic,
        2  ...,
        2  ...,
        2  * char(0) initial call plifill( addr(A), '00'X, stg(A) );
```

If the procedure invoked by the INITIAL CALL statement has been specified in a FETCH or RELEASE statement and it is not present in main storage, the INITIAL CALL statement initiates dynamic loading of the procedure. (For more information on dynamic loading, refer to "Dynamic loading of an external procedure" on page 101.)

Initializing array variables

Initial values specified for an array are assigned to successive elements of the array in row-major order (final subscript varying most rapidly). If too many initial values are specified, the excess values are ignored; if not enough are specified, the remainder of the array is not initialized.

The initialization of an array of strings can include both string repetition and iteration factors. Where only one of these is given, it is taken to be a string repetition factor unless the string constant is placed in parentheses.

The iteration factor may be specified as *, which means that all of the remaining elements will be initialized with the given value.

In the following examples:

`((2)'A')` is equivalent to `('AA')`

`((2)('A'))` is equivalent to `('A','A')`

`((2)(1)'A')` is equivalent to `('A','A')`

`((*)(1)'A')` is equivalent to `('A','A'...'A')`

An area variable is initialized with the value of the EMPTY built-in function, on allocation. Any INITIAL clause for an area variable will be ignored.

If the attributes of an item in the INITIAL attribute differ from those of the data item itself, conversion is performed, provided the attributes are compatible.

INITIAL is not allowed on objects of REFER clauses.

If the variable has the REFER option and the item involves a base element or a substructure or union of the current generation of the variable, the result of the

INITIAL attribute is undefined. In the following example the result of initializing D is undefined.

```
dcl 1 A based(Q),
      2 B fixed bin(15),
      2 C char(N refer(B))
             init('AAB'),
      2 D char(5) init(C);
allocate A;
```

Initializing unions

The members of a union can have initial values. However, if the union is static, only one member of the union can have the initial attribute. For nonstatic unions, initial attributes are applied in order of appearance. Subsequent initial values overwrite previous ones.

In the following example, the declaration for NT1 would be invalid if it had the static storage attribute. However, the declaration for NT2 is valid even though it has static storage class. Furthermore, the NT2 declaration is portable across EBCDIC and ASCII machines.

```
dcl
  1 NT1 union automatic,
    2 Numeric_translate_table1 char(256)
                  init( (256)'00'X),
    2 *,
      3 * char(240),
      3 * char(10) init('0123456789'),
    2 * char(0);

dcl
  1 NT2 union static,
    2 Numeric_translate_table2 char(256),
    2 *,
      3 * char(        index(collate(),'0')-1    )
          init((1)(low(index(collate(),'0')-1 )) ),
      3 * char(10) init('0123456789'),
      3 * char(         (256-(index(collate(),'0')-1)-10)    )
          init((1)(low( (256-(index(collate(),'0')-1)-10) )) ),
```

Initializing static variables

For a variable that is allocated when the program is loaded, that is, a static variable, which remains allocated throughout execution of the program, any value specified in an INITIAL attribute is assigned only once. (Static storage for fetched procedures is allocated and initialized each time the procedure is loaded.)

If static variables are initialized using the INITIAL attribute, the initial values must be specified as restricted expressions. Extent specifications must be restricted expressions.

The restrictions on initializing static variables are as follows:

- STATIC ENTRY variables must have the LIMITED attribute (see "LIMITED attribute" on page 117).
- INITIAL is not allowed for static format variables.
- INITIAL is allowed for label variables that are not part of structures or unions. In this case, the label variable gets the CONSTANT attribute.
- INITIAL is not allowed for static unaligned bit variables with inherited dimensions, unless initial values are all bit strings consisting solely of zeros.
- INITIAL is not valid for AREA variables.
- Only one element of a static union may specify INITIAL.
- If a STATIC EXTERNAL item without the RESERVED attribute is given the INITIAL attribute in more than one declaration, the value specified must be the same in every case.

Initializing automatic variables

For automatic variables, which are allocated at each activation of the declaring block, any specified initial value is assigned with each allocation.

Initializing based and controlled variables

For based and controlled variables, which are allocated at the execution of ALLOCATE statements (also LOCATE statements for based variables), any specified initial value is assigned with each allocation.

Examples

In the following example, when storage is allocated for `Name`, the character constant `'John Doe'` (padded on the right to 10 characters) is assigned to it.

```
dcl Name char(10) init('John Doe');
```

In the following example, when `Pi` is allocated, it is initialized to the value `3.1416`.

```
dcl Pi fixed dec(5,4) init(3.1416);
```

The following example specifies that `A` is to be initialized with the value of the expression `B*C`:

```
declare A init((B*C));
```

The following example results in each of the first 920 elements of `A` being set to `0`. The next 80 elements consist of 20 repetitions of the sequence `5,5,5,9`.

```
declare A (100,10) initial
        ((920)0, (20) ((3)5,9));
```

In the following example, only the first, third, and fourth elements of `A` are initialized; the rest of the array is not initialized. The array `B` is fully initialized, with the first 25 elements initialized to `0`, the next 25 to `1`, and the remaining elements to `0`. In the structure `C`, where the dimension `(8)` has been inherited by `D` and `E`, only the first element of `D` is initialized. All the elements of `E` are initialized.

```
declare A(15) character(13) initial
          ('John Doe',
           *,
           'Richard Row',
           'Mary Smith'),

        B (10,10) decimal fixed(5)
          init((25)0,(25)1,(*)0),

        1 C(8),
          2 D initial (0),
          2 E initial((*)0);
```

When an array of structures or unions is declared with the LIKE attribute to obtain the same structuring as a structure or union whose elements have been initialized, only the first structure or union is initialized.

In the following example only `J(1).H` and `J(1).I` are initialized in the array of structures.

```
declare 1 G,
          2 H initial(0),
          2 I initial(0),
        1 J(8) like G;
```

Chapter 10. Input and Output

Chapter 10. Input and output

PL/I input and output statements (such as READ, WRITE, GET, PUT) let you transmit data between the main and auxiliary storage of a computer. A collection of data external to a program is called a *data set.* Transmission of data from a data set to a program is called *input.* Transmission of data from a program to a data set is called *output.* (If you are using a terminal, "data set" can also mean your terminal.)

PL/I input and output statements are concerned with the logical organization of a data set and not with its physical characteristics. A program can be designed without specific knowledge of the input/output devices that will be used when the program is executed. To allow a source program to deal primarily with the logical aspects of data rather than with its physical organization in a data set, PL/I employs models of data sets, called *files.* A file can be associated with different data sets at different times during the execution of a program.

PL/I uses two types of data transmission: stream and record.

In stream-oriented data transmission, the organization of the data in the data set is ignored within the program, and the data is treated as though it were a continuous stream of individual data values in character form. Data is converted from character form to internal form on input, and from internal form to character form on output.

For more information on stream-oriented data transmission, refer to Chapter 12, "Stream-oriented data transmission" on page 252

Stream-oriented data transmission can be used for processing input data prepared in character form and for producing readable output, where editing is required. Stream-oriented data transmission allows synchronized communication with the program at run time from a terminal, if the program is interactive.

Stream-oriented data transmission is more versatile than record-oriented data transmission in its data-formatting abilities, but is less efficient in terms of run time.

In record-oriented data transmission, the data set is a collection of discrete records. The record on the external medium is generally an exact copy of the record as it exists in internal storage. No data conversion takes place during record-oriented data transmission. On input the data is transmitted exactly as it is recorded in the data set, and on output it is transmitted exactly as it is recorded internally.

For more information on record-oriented data transmission, refer to Chapter 11, "Record-oriented data transmission" on page 242

Record-oriented data transmission can be used for processing files that contain data in any representation, such as binary, decimal, or character.

Record-oriented data transmission is more versatile than stream-oriented data transmission, in both the manner in which data can be processed and the types of data sets that it can process. Since data is recorded in a data set exactly as it appears in main storage, any data type is acceptable, No conversions occur, but you must have a greater awareness of the data structure.

It is possible for the same data set to be processed at different times by either stream or record data transmission. However, all items in the data set must be in character form.

The following sections in this chapter discuss the kinds of data sets, the attributes for describing files, and how you open and close files in order to transmit data. For more information about the types of data set organizations that PL/I recognizes, refer to *PL/I Package/2 Programming Guide*.

Data sets

In addition to being used as input from and output to your terminal, data sets are stored on a variety of auxiliary storage media, including magnetic tape and direct-access storage devices (DASDs). Despite their variety, these media have characteristics that allow common methods of collecting, storing, and transmitting data. The organization of a data set determines how data is recorded in a data set and how the data is subsequently retrieved so that it can be transmitted to the program. Records are stored in and retrieved from a data set either sequentially on the basis of successive physical or logical positions, or directly by the use of keys specified in data transmission statements.

PL/I supports the following types of data set organizations:

Consecutive
Indexed
Relative
Regional

The data set organizations differ in the way they store data and in the means they use to access data.

Consecutive

In the consecutive data set organization, records are organized solely on the basis of their successive physical positions. When the data set is created, records are written consecutively in the order in which they are presented. The records can be retrieved only in the order in which they were written.

Indexed

In the indexed data set organization, records are placed in a logical sequence based on the key of each record. An indexed data set must reside on a direct-access device. A character string key identifies the record and allows direct retrieval, replacement, addition, and deletion of records. Sequential processing is also allowed.

Relative

In the relative data set organization, numbered records are placed in a position relative to each other. The records are numbered in succession, beginning with one. A relative data set must reside on a direct-access device. A key that specifies the record number identifies the record and allows direct retrieval, replacement, addition, and deletion of records. Sequential processing is also allowed.

Regional

The regional data set organization is divided into numbered regions, each of which can contain one record. The regions are numbered in succession, beginning with zero. A region can be accessed by specifying its region number, and perhaps a key, in a data transmission statement. The key specifies the region number and identifies the region to allow optimized direct retrieval, replacement, addition, and deletion of records.

Files

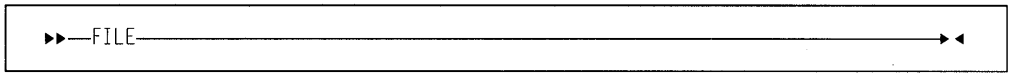
To allow a source program to deal primarily with the logical aspects of data rather than with its physical organization in a data set, PL/I employs models of data sets, called *files*. These models determine how input and output statements access and process the associated data set. Unlike a data set, a file data item has significance only within the source program and does not exist as a physical entity external to the program.

A name that represents a file has the FILE attribute.

FILE attribute

The FILE attribute specifies that the associated name is a file constant or file variable.

The syntax for the FILE attribute is:



The FILE attribute can be implied for a file constant by any of the file description attributes. A name can be contextually declared as a file constant through its appearance in the FILE option of any input or output statement, or in an ON statement for any input/output condition.

File constant

Each data set processed by a PL/I program must be associated with a file constant.

The individual characteristics of each file constant are described with file description attributes. These attributes fall into two categories: alternative attributes and additive attributes.

An alternative attribute is one that is chosen from a group of attributes. If no explicit or implied attribute is given for one of the alternatives in a group and if one of the alternatives is required, a default attribute is used.

Figure 39 lists the PL/I alternative file attributes.

Figure 39 (Page 1 of 2). Alternative file attributes

Group Type	Alternative Attributes	Default Attribute
Usage	STREAM or RECORD	STREAM
Function	INPUT or OUTPUT or UPDATE	INPUT

Figure 39 (Page 2 of 2). Alternative file attributes

Group Type	Alternative Attributes	Default Attribute
Access	SEQUENTIAL or DIRECT	SEQUENTIAL
Buffering	BUFFERED or UNBUFFERED	BUFFERED (for SEQUENTIAL files); UNBUFFERED (for DIRECT files)
Scope	EXTERNAL or INTERNAL	EXTERNAL

An additive attribute is one that must be stated explicitly or is implied by another explicitly stated attribute. The additive attributes are ENVIRONMENT, KEYED and PRINT. The additive attribute KEYED is implied by the DIRECT attribute. The additive attribute PRINT can be implied by the output file name SYSPRINT.

Figure 40 show the attributes that apply to each type of data transmission:

Figure 40. Attributes by data transmission type

Type of transmission	Attribute
Stream-oriented	ENVIRONMENT
	INPUT and OUTPUT
	PRINT
	STREAM
Record-oriented	BUFFERED and UNBUFFERED
	DIRECT and SEQUENTIAL
	ENVIRONMENT
	INPUT, OUTPUT, and UPDATE
	KEYED
	RECORD

Figure 41 shows the valid combinations of file attributes.

Figure 41 (Page 1 of 2). Attributes of PL/I file declarations

File Type	STREAM	RECORD					Legend: I Must be specified or implied; D Default; O Optional; S Must be specified; - Invalid
		SEQUENTIAL			DIRECT		
Data Set Organization	Consecutive	Consecutive	Relative	Indexed	Relative	Indexed	
File Attributes							**Attributes Implied**
FILE	I	I	I	I	I	I	
INPUT[1]	D	D	D	D	D	D	FILE
OUTPUT	O	O	O	O	O	O	FILE
ENVIRONMENT	O	O	O	O	O	O	FILE
STREAM	D	-	-	-	-	-	FILE
PRINT[1]	O	-	-	-	-	-	FILE STREAM OUTPUT
RECORD	-	I	I	I	I	I	FILE
UPDATE	-	O	O	O	O	O	FILE RECORD

Figure 41 (Page 2 of 2). Attributes of PL/I file declarations

File Type	STREAM	RECORD					Legend:
		SEQUENTIAL			DIRECT		I Must be specified or implied D Default O Optional S Must be specified - Invalid
Data Set Organization	Consecutive	Consecutive	Relative	Indexed	Relative	Indexed	
SEQUENTIAL	-	D	D	D	-	-	FILE RECORD
KEYED[2]	-	-	O	O	I	I	FILE RECORD
DIRECT	-	-	-	-	S	S	FILE RECORD KEYED

Notes:

[1] A file with the INPUT attribute cannot have the PRINT attribute
[2] KEYED is required for *indexed* and *relative* output

Scope is discussed in "Scope of declarations" on page 132.

The FILE attribute can be implied for a file constant by any of the file description attributes discussed in this chapter. A name can be contextually declared as a file constant through its appearance in the FILE option of any input or output statement, or in an ON statement for any input/output condition.

In the following example, the name `Master` is declared as a file constant:

```
declare Master file;
```

File variable

A file variable has the attributes FILE and VARIABLE. It cannot have any of the file constant description attributes. File constants can be assigned to file variables. After assignment, a reference to the file variable has the same significance as a reference to the assigned file constant.

The value of a file variable can be transmitted by record-oriented transmission statements. The value of the file variable on the data set might not be valid after transmission.

The VARIABLE attribute is implied under the circumstances described in "VARIABLE attribute" on page 48.

In the following declaration `Account` is declared as a file variable, and `Acct1` and `Acct2` are declared as file constants. The file constants can subsequently be assigned to the file variable.

```
declare Account file variable,
  Acct1 file,
  Acc2 file;
```

For syntax information, refer to "VARIABLE attribute" on page 48.

Specifying a file reference

A file reference can be a file constant, a file variable, or a function reference which returns a value with the FILE attribute. It can be used in the following ways:

- In a FILE or COPY option
- As an argument to be passed to a function or subroutine
- To qualify an input/output condition for ON, SIGNAL, and REVERT statements
- As the expression in a RETURN statement.

On-units can be established for a file constant through a file variable that represents its value (see "ON-units for file variables" on page 307). In the following example, the statements labelled L1 and L2 both specify null ON-units for the same file.

```
dcl F file,
    G file variable;
    G=F;
L1:  on endfile(G);
L2:  on endfile(F);
```

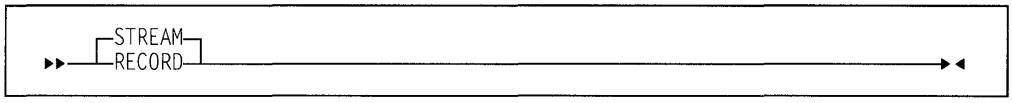
RECORD and STREAM attributes

The RECORD and STREAM usage attributes specify the kind of data transmission used for the file.

RECORD indicates that the file consists of a collection of physically separate records, each of which consists of one or more data items in any form. Each record is transmitted as an entity to or from a variable.

STREAM indicates that the data of the file is a continuous stream of data items, in character form, assigned from the stream to variables, or from expressions into the stream.

The syntax for the RECORD and STREAM attributes is:



Default: STREAM.

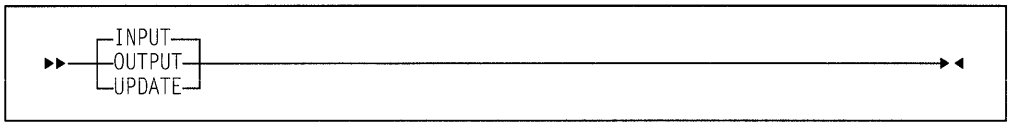
A file with the STREAM attribute can be specified only in the FILE option of the OPEN, CLOSE, GET, and PUT input/output statements.

A file with the RECORD attribute can be specified only in the FILE option of the OPEN, CLOSE, READ, WRITE, REWRITE, LOCATE, and DELETE input/output statements.

INPUT, OUTPUT, and UPDATE attributes

The INPUT, OUTPUT and UPDATE function attributes specify the direction of data transmission allowed for a file. INPUT specifies that data is transmitted from a data set to the program. OUTPUT specifies that data is transmitted from the program to a data set, either to create a new data set or to extend an existing one. UPDATE, which applies to RECORD files only, specifies that the data can be transmitted in either direction. A declaration of UPDATE for a SEQUENTIAL file indicates the update-in-place mode.

The syntax for the INPUT, OUTPUT, and UPDATE attributes is:

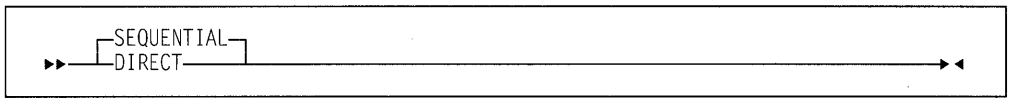


Default: INPUT.

SEQUENTIAL and DIRECT attributes

The SEQUENTIAL and DIRECT access attributes apply only to RECORD files, and specify how the records in the file are accessed.

The syntax for the SEQUENTIAL and DIRECT attributes is:



Abbreviation: SEQL for SEQUENTIAL

Default: SEQUENTIAL.

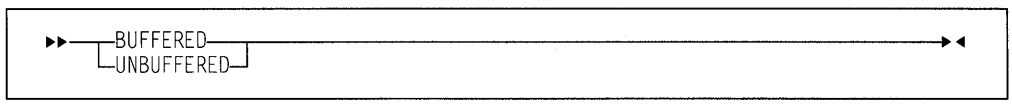
The DIRECT attribute specifies that records in a data set are directly accessed. The location of the record in the data set is determined by a character-string key. Therefore, the DIRECT attribute implies the KEYED attribute. The associated data set must be on a direct-access storage device.

The SEQUENTIAL attribute specifies that records in a consecutive or relative data set are accessed in physical sequence, and that records in an indexed data set are accessed in key sequence order. For certain data set organizations, a file with the SEQUENTIAL attribute can also be used for direct access or for a mixture of random and sequential access. In this case, the file must have the additive attribute KEYED. Existing records of a data set in a SEQUENTIAL UPDATE file can be modified, ignored, or, if the data set is indexed, deleted.

BUFFERED and UNBUFFERED attributes

The buffering attributes apply only to RECORD files.

The syntax for the BUFFERED and UNBUFFERED attributes is:



Abbreviations: BUF for BUFFERED, and UNBUF for UNBUFFERED

Defaults: BUFFERED is the default for SEQUENTIAL files. UNBUFFERED is the default for DIRECT files.

The BUFFERED attribute specifies that during transmission to and from a data set, each record of a RECORD file must pass through intermediate storage buffers. This allows both move and locate mode processing.

The UNBUFFERED attribute indicates that a record in a data set need not pass through a buffer but can be transmitted directly to and from the main storage associated with a variable. This allows only move mode processing.

ENVIRONMENT attribute

The characteristic list of the ENVIRONMENT attribute specifies various data set characteristics that are not part of PL/I. For a full description of the characteristics and their uses, refer to the *PL/I Package/2 Programming Guide*

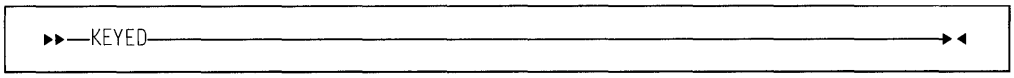
Note: Because the characteristics are not part of the PL/I language, using them in a file declaration can limit the portability of your application program.

The following characteristics can be specified on the ENVIRONMENT attribute. For descriptions of the characteristics, refer to *PL/I Package/2 Programming Guide*.

BKWD	GRAPHIC	RECSIZE
CONSECUTIVE	KEYLENGTH	REGIONAL(1)
CTLASA	KEYLOC	SCALARVARYING
GENKEY	ORGANIZATION	VSAM

KEYED attribute

The KEYED attribute applies only to RECORD files, and must be associated with indexed and relative data sets. It specifies that records in the file can be accessed using one of the key options (KEY, KEYTO, or KEYFROM) of record I/O statements. The syntax for the KEYED attribute is:



The KEYED attribute need not be specified unless one of the key options is used.

PRINT attribute

The PRINT attribute is described in "PRINT attribute" on page 270.

Opening and closing files

Before a file can be used for data transmission, by input or output statements, it must be associated with a data set. Opening a file associates the file with a data set and involves checking for the availability of external media, positioning the media, and allocating required operating system support. When processing is completed, the file must be closed. Closing a file dissociates the file from the data set.

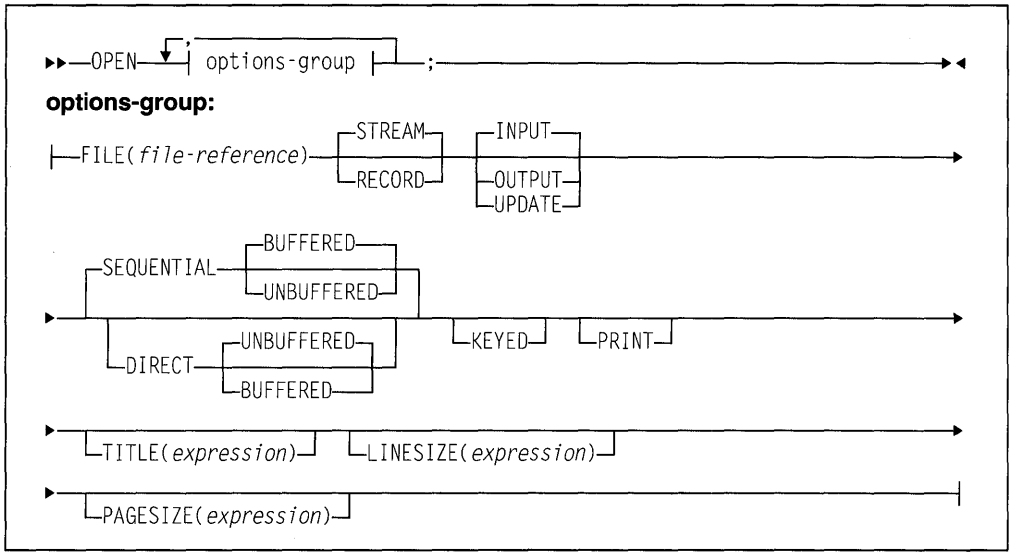
PL/I provides two statements, OPEN and CLOSE, to perform these functions. However, use of these statements is optional. If an OPEN statement is not executed for a file, the file is opened implicitly during the execution of first data transmission statement for that file. In this case, the file opening uses information about the file as specified in a DECLARE statement (and defaults derived from the transmission statement). Similarly, if a file has not been closed before PL/I termination, PL/I will close it during the termination process.

When a file for stream input, sequential input, or sequential update is opened, the associated data set is positioned at the first record.

OPEN statement

The OPEN statement associates a file with a data set. It merges attributes specified on the OPEN statement with those specified on the DECLARE statement. It also completes the specification of attributes for the file, if a complete set of attributes has not been declared for the file being opened.

The syntax for the OPEN statement is:



The options of the OPEN statement can appear in any order.

FILE specifies the name of the file that is to be associated with a data set.

STREAM, RECORD,
INPUT, OUTPUT, UPDATE,
DIRECT, SEQUENTIAL,
BUFFERED, UNBUFFERED,
KEYED, and PRINT

options specify attributes that augment the attributes specified in the file declaration. The same attributes need not be listed in both OPEN and DECLARE statements for the same file. For a list of attributes for record and stream input and output, see Figure 40 on page 231.

When a STREAM file is opened, the first GET or PUT statement can specify, with a statement option or format item, the first record to be accessed. The statement option or format item indicates that *n* lines are skipped before a record is accessed. The file is then positioned at the start of the *n*th record. If no statement option or format item is encountered, the initial file position is the start of the first line or record. If the file has the PRINT attribute, it is physically positioned at column 1 of the first line or record

Opening a file that is already open does not affect the file.

TITLE The content of *expression* determines what is being designated. For more information on the TITLE attribute refer to *PL/I Package/2 Programming Guide*.

LINESIZE

converted to an integer value, specifies the length in bytes of a line during subsequent operations on the file. New lines can be started by use of the printing and control format items or by options in a GET or PUT statement. If an attempt is made to position a file past the end of a line before explicit action to start a new line is taken, a new line is started, and the file is positioned to the start of this new line. The default line size for PRINT file is 120.

The LINESIZE option can be specified only for a STREAM OUTPUT file. The line size taken into consideration whenever a SKIP option appears in a GET statement is the line size that was used to create the data set. Otherwise, the line size is taken as the current length of the logical record minus control bytes.

PAGESIZE

is evaluated and converted to an integer value, and specifies the number of lines per page. The first attempt to exceed this limit raises the ENDPAGE condition. During subsequent transmission to the PRINT file, a new page can be started by use of the PAGE format item or by the PAGE option in the PUT statement. The default page size is 60.

The PAGESIZE option can be specified only for a file having the PRINT attribute.

Implicit opening

An implicit opening of a file occurs when a GET, PUT, READ, WRITE, LOCATE, REWRITE, or DELETE statement is executed for a file for which an OPEN statement has not already been executed.

If a GET statement contains a COPY option, execution of the GET statement can cause implicit opening of either the file specified in the COPY option or, if no file was specified, of the output file SYSPRINT. Implicit opening of the file specified in the COPY option implies the STREAM and OUTPUT attributes.

Figure 42 shows the attributes that are implied when a given statement causes the file to be implicitly opened :

Figure 42. Attributes implied by implicit open

Statement	Implied Attributes
GET	STREAM, INPUT
PUT	STREAM, OUTPUT
READ	RECORD, INPUT(Note 1)
WRITE	RECORD, OUTPUT(Note 1)
LOCATE	RECORD, OUTPUT, SEQUENTIAL
REWRITE	RECORD, UPDATE
DELETE	RECORD, UPDATE

Note 1:

INPUT and OUTPUT are default attributes for READ and WRITE statements only if UPDATE has not been explicitly declared.

When one of the statements listed in Figure 42 opens a file implicitly, it is functionally equivalent to using an explicit OPEN statement for the file with the same attributes specified.

There must be no conflict between the attributes specified in a file declaration and the attributes implied as the result of opening the file. For example, the attributes INPUT and UPDATE are in conflict, as are the attributes UPDATE and STREAM.

The implied attributes discussed earlier are applied before the default attributes listed in Figure 42 on page 237 are applied. Implied attributes can also cause a conflict. If a conflict in attributes exists after the application of default attributes, the UNDEFINEDFILE condition is raised.

Figure 43. Merged and implied attributes

Merged Attributes	Implied Attributes
UPDATE	RECORD
SEQUENTIAL	RECORD
DIRECT	RECORD, KEYED
PRINT	OUTPUT, STREAM
KEYED	RECORD

The following two examples illustrate attribute merging for an explicit opening using a file constant and a file variable:

Example of file constant

```
declare Listing file stream;
open file(Listing) print;
```

Attributes after merge caused by execution of the OPEN statement are STREAM and PRINT. Attributes after implication are STREAM, PRINT, and OUTPUT. Attributes after default application are STREAM, PRINT, OUTPUT, and EXTERNAL.

Example of file variable

```
declare Account file variable,
        (Acct1,Acct2)  file
        output;

Account = Acct1;
open file(Account) print;

Account = Acct2;
open file(Account) record unbuf;
```

The file `Acct1` is opened with attributes (explicit and implied) STREAM, EXTERNAL, PRINT, and OUTPUT. The file `Acct2` is opened with attributes RECORD, EXTERNAL, OUTPUT,

Example of implicit opening

```
declare Master file keyed internal;

real file (Master)
     into (Master_Record)
     keyto(Master_Key);
```

Attributes after merge (from the implicit opening caused by execution of the READ statement) are KEYED, INTERNAL, RECORD, and INPUT. (No additional attributes are implied.) Attributes after default application are KEYED, INTERNAL, RECORD, INPUT, and SEQUENTIAL.

Examples of declarations of file constants

```
declare File3 input direct environment( regional(1) )
```

This declaration specifies three file attributes: INPUT, DIRECT, and ENVIRONMENT. Other implied attributes are FILE (implied by each of the attributes) and RECORD and KEYED (implied by DIRECT). Scope is EXTERNAL, by default. The ENVIRONMENT attributes specifies that the data set is of the REGIONAL(1) organization.

For the previous declaration, all necessary attributes are either stated or implied in the DECLARE statement. None of the stated attributes can be changed (or overridden) in an OPEN statement.

If the declaration is written as shown in the following example, `invntry` can be opened for different purposes.

```
declare invntry file;
```

In the following example, the file attributes are the same as those specified (or implied) in the DECLARE statement in the previous example.

```
open file (Invntry)
     update sequential;
```

The file might be opened in this way, then closed, and then later opened with a different set of attributes. For example, the following OPEN statement allows records to be read with either the KEYTO or the KEY option.

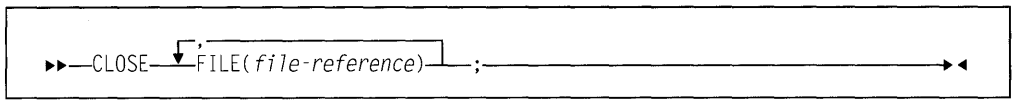
```
open file (Invntry)
     input sequential keyed;
```

Because the file is SEQUENTIAL, the data set can be accessed in a purely sequential manner. It can also be accessed directly by means of a READ statement with a KEY option. A READ statement with a KEY option for a file of this description obtains a specified record. Subsequent READ statements without a KEY option access records sequentially, beginning with the next record in KEY sequence.

CLOSE statement

The CLOSE statement dissociates an opened file from its data set.

The syntax for the CLOSE statement is:



FILE specifies the name of the file that is to be dissociated from the data set.

The CLOSE statement also dissociates from the file all attributes established for it by the implicit or explicit opening process. If desired, new attributes can be speci-

fied for the file in a subsequent OPEN statement. However, all attributes explicitly given to the file constant in a DECLARE statement remain in effect.

Closing a file that was previously closed has no effect. A closed file can be reopened. If a file is not closed by a CLOSE statement, it is closed at the termination of the program.

Chapter 11. Record-oriented data transmission

Chapter 11. Record-oriented data transmission

This chapter describes features of the input and output statements used in record-oriented data transmission. Those features of PL/I that apply generally to record-oriented or stream-oriented data transmission, including declaring files, file attributes, and opening and closing files, are described in Chapter 10, "Input and output." For syntax information about the ENVIRONMENT attribute refer to "ENVIRONMENT attribute" on page 235. For details about environment characteristics and record I/O data transmission statements for each data set organization refer to the *PL/I Package/2 Programming Guide*.

In record-oriented data transmission, data in a data set is a collection of records recorded in any format acceptable to the operating system. No data conversion is performed during record-oriented data transmission. On input, the READ statement either transmits a single record to a program variable exactly as it is recorded in the data set, or sets a pointer to the record. On output, the WRITE, REWRITE, or LOCATE statement transmits a single record from a program variable exactly as it is recorded internally. If the information transmitted to the file has a length N which is less than the established record length M, the resulting value of the last M-N bytes of the record is undefined.

Data transmitted

Most variables, including parameters and DEFINED variables, can be transmitted by record-oriented data transmission statements. In general, the information given in this chapter can be applied equally to all variables.

Note: A data aggregate must be in connected storage. If a graphic string is specified for input or output, the SCALARVARYING option must be specified for the file. Other data considerations are described in the following sections.

Unaligned bit strings

The following cannot be transmitted:

- BASED, DEFINED, parameter, subscripted, or structure-base-element variables that are unaligned nonvarying bit strings
- Minor structures whose first or last base elements are unaligned nonvarying bit strings (except where they are also the first or last elements of the containing major structure)
- Major structures that have the DEFINED attribute or are parameters, and that have unaligned nonvarying bit strings as their first or last elements.

VARYING strings

A locate mode output statement (see "LOCATE statement" on page 245) specifying a VARYING string transmits a field having a length equal to the maximum length of the string, plus a 2-byte prefix denoting the current length of the string. The SCALARVARYING option of the ENVIRONMENT attribute must be specified for the file.

A move mode output statement (see "WRITE statement" on page 244 and "REWRITE statement" on page 244) specifying a VARYING string variable trans-

mits only the current length of the string. A 2-byte prefix is included only if the SCALARVARYING option of the ENVIRONMENT attribute is specified for the file.

Reading and writing using varying strings allows you to access records that may have undefined or unknown lengths.

Area variables

A locate mode output statement specifying an area variable transmits a field whose length is the declared size of the area, plus a 16-byte prefix containing control information.

A move mode statement specifying an element area variable or a structure whose last element is an area variable transmits only the current extent of the area plus a 16-byte prefix.

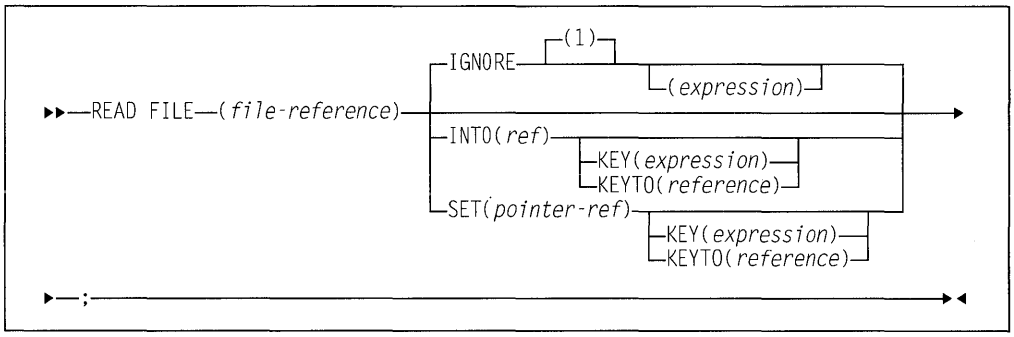
Data transmission statements

The data transmission statements that transmit records to or from a data set are READ, WRITE, LOCATE, and REWRITE. The DELETE statement deletes records from an UPDATE file. The attributes of the file determine which data transmission statements can be used. Statement options are described in "Options of data transmission statements" on page 245. For information about variables in data transmission statements, see the *PL/I Package/2 Programming Guide*.

READ statement

The READ statement can be used with any INPUT or UPDATE file. It either transmits a record from the data set to the program variable or sets a pointer to the record in storage.

The syntax for the READ statement is:

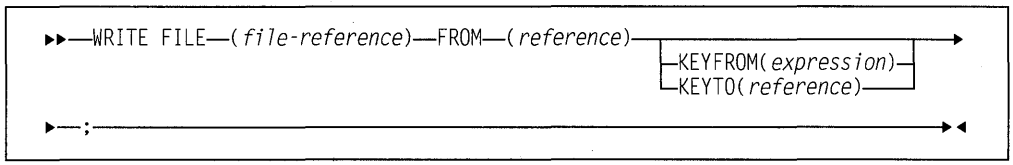


The keywords can appear in any order. A READ statement without an INTO, SET, or IGNORE option is equivalent to a READ with an IGNORE(1).

WRITE statement

The WRITE statement can be used with any OUTPUT file, DIRECT UPDATE file, or SEQUENTIAL UPDATE file. It transmits a record from the program and adds it to the data set.

The syntax for the WRITE statement is:

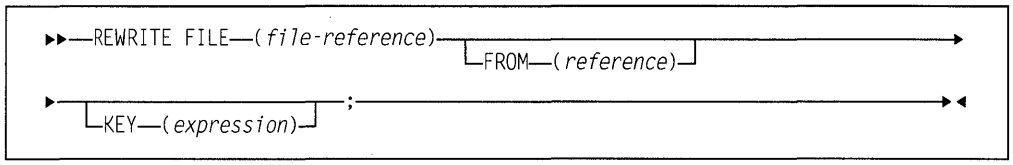


The keywords can appear in any order.

REWRITE statement

The REWRITE statement replaces a record in an UPDATE file. For SEQUENTIAL UPDATE files, the REWRITE statement specifies that the last record read from the file is to be rewritten; consequently a record must be read before it can be rewritten. For DIRECT UPDATE files, any record can be rewritten whether or not it has first been read.

The syntax for the REWRITE statement is:



The keywords can appear in any order. The FROM option must be specified for UPDATE files with the DIRECT attribute, or with both the SEQUENTIAL and UNBUFFERED attributes.

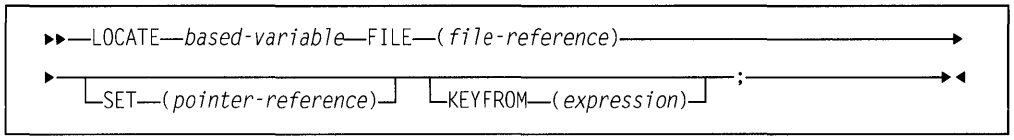
A REWRITE statement that does not specify the FROM option has the following effect:

- If the last record was read by a READ statement with the INTO option, REWRITE without FROM has no effect on the record in the data set.
- If the last record was read by a READ statement with the SET option, the record is updated by whatever assignments were made in the variable identified by the pointer variable in the SET option.

LOCATE statement

The LOCATE statement can be used only with an OUTPUT SEQUENTIAL BUFFERED file for locate mode processing. It allocates storage within an output buffer for a based variable and sets a pointer to the location of the next record. For further description of locate mode processing, see "Locate mode" on page 249.

The syntax for the LOCATE statement is:



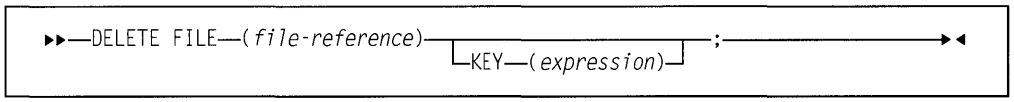
The keywords can appear in any order.

based-variable
must be an unsubscripted level-1 based variable.

DELETE statement

The DELETE statement deletes a record from an UPDATE file.

The syntax for the DELETE statement is:



The keywords can appear in any order. If the KEY option is omitted, the record to be deleted is the last record that is read. No subsequent DELETE or REWRITE statement without a KEY is allowed until another READ statement is processed. If the KEY option is included, that record addressed by the key will be deleted if found.

Options of data transmission statements

Options that are allowed for record-oriented data transmission statements differ according to the attributes of the file and the characteristics of the associated data set.

FILE option

The FILE option must appear in every record-oriented data transmission statement. It specifies the file upon which the operation takes place. An example of the FILE option is shown in each of the statements in this section. If the file specified is not open in the current process, it is opened implicitly.

FROM option

The FROM option specifies the element or aggregate variable from which the record is written. The FROM option must be used in the WRITE statement for any OUTPUT or DIRECT UPDATE file. It can also be used in the REWRITE statement for any UPDATE file.

If the variable is an aggregate, it must be in connected storage. Certain uses of unaligned nonvarying bit strings are disallowed (for details, see "Data transmitted" on page 242).

The FROM variable can be an element string variable of varying length. When using a WRITE statement with the FROM option, only the current length of a VARYING string is transmitted to a data set, and a 2-byte prefix specifying the length can be attached. It is attached only if the SCALARVARYING option of the ENVIRONMENT attribute is specified for the file.

Records are transmitted as an integral number of bytes. If a bit string (or a structure that starts or ends with a bit string) that is not aligned on a byte boundary is transmitted, the record is padded with bits at the start or the end of the string, and the result might be incorrect.

The FROM option can be omitted from a REWRITE statement for SEQUENTIAL UPDATE files. If the last record was read by a READ statement with the INTO option, REWRITE without FROM has no effect on the record in the data set. If the last record was read by a READ statement with the SET option, the record (updated by whatever assignments were made) is copied back onto the data set.

In the following examples, the statements specify that the value of the variable `Mas_Rec` is written into the output file `Master`.

```
write file (Master) from (Mas_Rec);
```

The REWRITE statement specifies that `Mas_Rec` replaces the last record read from an UPDATE file.

```
rewrite file (Master) from (Mas_Rec);
```

IGNORE option

The IGNORE option can be used in a READ statement for any SEQUENTIAL INPUT or SEQUENTIAL UPDATE file.

The expression in the IGNORE option is evaluated and converted to an integer value *n*. If *n* is greater than zero, *n* records are ignored. A subsequent READ statement for the file will access the (*n*+1)th record. If *n* is less than 1, the READ statement has no effect.

The following example specifies that the next three records in the file are to be ignored:

```
real file (In) ignore (3);
```

INTO option

The INTO option specifies an element or aggregate variable into which the logical record is read. The INTO option can be used in the READ statement for any INPUT or UPDATE file.

If the variable is an aggregate, it must be in connected storage. Certain uses of unaligned nonvarying bit strings are disallowed (for details, see "Data transmitted" on page 242).

The INTO variable can be an element string variable of varying length. If the SCALARVARYING option of the ENVIRONMENT attribute was specified for the file, each record contains a 2-byte prefix that specifies the length of the string data.

If SCALARVARYING was not declared then, on input, the string length is calculated from the record length and attached as a 2-byte prefix. For VARYING bit strings, this calculation rounds up the length to a multiple of 8 and therefore the calculated length might be greater than the maximum declared length.

The following example specifies that the next sequential record is read into the variable `RECORD_1`:

```
read file (Detail) into (Record_1);
```

KEY option

The KEY option specifies a character or graphic key that identifies a record. It can be used in a READ statement for an INPUT or UPDATE file, or in a REWRITE statement for a DIRECT UPDATE file.

The KEY option applies only to KEYED files. The KEY option is required if the file has the DIRECT attribute and optional if the file has the SEQUENTIAL and KEYED attributes.

The expression in the KEY option is evaluated and, if not character or graphic, is converted to a character value that represents a key. It is this character or graphic value that determines which record is read.

The following example specifies that the record identified by the character value of the variable `Stkey` is read into the variable `Item`:

```
read file (Stpck) into (Item) key (Stkey);
```

KEYFROM option

The KEYFROM option specifies a character or graphic key that identifies the record on the data set to which the record is transmitted. It can be used in a WRITE statement for any KEYED OUTPUT or DIRECT UPDATE file, or in a LOCATE statement.

The KEYFROM option applies only to KEYED files. The expression is evaluated and, if not character or graphic, is converted to a character string and is used as the key of the record when it is written.

Relative data sets can be created using the KEYFROM option. The record number is specified as the key.

REGIONAL(1) data sets can be created using the KEYFROM option. The region number is specified as the key.

For indexed data sets, KEYFROM specifies a recorded key whose length must be equal to the key length specified for the data set.

The following example specifies that the value of Loanrec is written as a record in the file Loans, and that the character string value of Loanno is used as the key with which it can be retrieved:

```
write file (Loans) from (Loanrec) keyfrom (Loanno);
```

KEYTO option

The KEYTO option specifies the character or graphic variable to which the key of a record is assigned. The KEYTO option can specify any string pseudovariable other than STRING. It cannot specify a variable declared with a numeric picture specification. The KEYTO option can be used in a READ statement for a SEQUENTIAL INPUT or SEQUENTIAL UPDATE file.

The KEYTO option applies only to KEYED files.

Assignment to the KEYTO variable always follows assignment to the INTO variable. If an incorrect key specification is detected, the KEY condition is raised. The value assigned is as follows:

- For indexed data sets, the record key is padded or truncated on the right to the declared length of the character variable.
- For *relative* data sets, a record number is converted to a character string with leading zeros suppressed, truncated, or padded on the left to the declared length of the character variable.
- For REGIONAL(1) data sets, the 9-character region-number, padded or truncated on the left to the declared length of the character variable. If the character variable is of varying length, any leading zeros in the region number are truncated and the string length is set to the number of significant digits. An all-zero region number is truncated to a single zero.

The KEY condition is not raised for this type of padding or truncation.

The following example specifies that the next record in the file Detail is read into the variable Invntry, and that the key of the record is assigned to the variable Keyfld:

```
read file (Detail) into (Invntry) keyto (Keyfld);
```

SET option

The SET option can be used with a READ statement or a LOCATE statement. For the READ statement, it specifies a pointer variable that is set to point to the record read. For the LOCATE statement, it specifies a pointer variable that is set to point to the next record for output.

If the SET option is omitted for the LOCATE statement, the pointer declared with the record variable is set. If a VARYING string is transmitted, the SCALARVARYING option must be specified for the file.

The following example specifies that the value of the pointer variable P is set to the location in the buffer of the next sequential record:

```
read file (X) set (P);
```

Processing modes

Record-oriented data transmission has two modes of handling data:

Move mode processes data by moving it into or out of the variable.

Locate mode processes data while it remains in a buffer. The execution of a data transmission statement assigns a pointer variable for the location of the storage allocated to a record in the buffer. Locate mode is applicable only to BUFFERED files.

The data transmission statements and options that you specify determine the processing mode used.

Move mode

In move mode, a READ statement transfers a record from the data set to the variable named in the INTO option. A WRITE or REWRITE statement transfers a record from the variable named in the FROM option to the data set. The variables named in the INTO and FROM options can be of any storage class.

The following is an example of move mode input:

```
Eof_In = '0'b;
on endfile(In) Eof_In = '1'B;
read file(In) into(Data);
do while (¬Eof_In);
   ⋮
         /* process record  */
   read file(In) into(Data);
end;
```

Locate mode

Locate mode assigns to a pointer variable the location of the buffer. A based variable described the record. The same data can be interpreted in different ways by using different based variables. Locate mode can also be used to read self-defining records, in which information in one part of the record is used to indicate the structure of the rest of the record. For example, this information could be an array bound or a code identifying which based structure should be used for the attributes of the data.

A READ statement with a SET option sets the pointer variable in the SET option to a buffer containing the record. The data in the record can then be referenced by a based variable qualified with the pointer variable.

The pointer value is valid only until the execution of the next READ or CLOSE statement that refers to the same file.

The pointer variable specified in the SET option or, if SET was omitted, the pointer variable specified in the declaration of the based variable, is used. The pointer value is valid only until the execution of the next LOCATE, WRITE, or CLOSE statement that refers to the same file. It also initializes components of the based variable that have been specified in REFER options.

The LOCATE statement sets a pointer variable to a large enough area where the next record can be built.

After execution of the LOCATE statement, values can be assigned directly into the based variables qualified by the pointer variable set by the LOCATE statement.

The following example shows locate mode input:

```
dcl 1 Data based(P),
      2
      ⋮
      ;

on endfile(In)
;
read file(In) set(P);
do while (¬endfile(In));
   ⋮
                        /* process record */
   read file(In) set(P);
end;
```

The following example shows locate mode output:

```
dcl 1 Data based(P);
       2
      ⋮
      ;

do while (More_records_to_write);
   locate Data file(Out);
   ⋮
               /* build record */
end;
```

Chapter 12. Stream-oriented data transmission

Chapter 12. Stream-oriented data transmission

This chapter describes the input and output statements used in stream-oriented data transmission. Features that apply to stream-oriented and record-oriented data transmission, including files, file attributes, and opening and closing files, are described in Chapter 10, "Input and output" on page 228.

Stream-oriented data transmission treats a data set as a continuous stream of data values in character, graphic, or mixed character data form. Within a program, record boundaries are generally ignored. However, a data set consists of a series of lines of data, and each data set created or accessed by stream-oriented data transmission has a line size associated with it. In general, a line is equivalent to a record in the data set, but the line size does not necessarily equal the record size.

The stream-oriented data transmission statements can also be used for internal data movement, by specifying the STRING option instead of specifying the FILE option. Although the STRING option is not an input/output operation, its use is described in this chapter.

Stream-oriented data transmission can be list-directed, data-directed, or edit-directed.

List-directed data transmission
transmits the values of data-list items without your having to specify the format of the values in the stream. The values are recorded externally as a list of constants, separated by blanks or commas.

Data-directed data transmission
transmits the names of the data-list items, as well as their values, without your having to specify the format of the values in the stream. The GRAPHIC option of the ENVIRONMENT attribute must be specified if any variable has a DBCS name, even if no DBCS data is present.

Edit-directed data transmission
transmits the values of data-list items and requires that you specify the format of the values in the stream. The values are recorded externally as a string of characters or graphics to be treated character by character (or graphic by graphic) according to a format list.

The following sections detail the data transmission statements and their options, and how to specify the list-, data-, and edit-directed data. How to accommodate double-byte characters is discussed in "DBCS data in stream I/O" on page 272.

Data transmission statements

Stream-oriented data transmission uses GET and PUT statements. Only consecutive files can be processed with the GET and PUT statements.

The variables or pseudovariables to which data values are assigned, and the expressions from which they are transmitted, are generally specified in a data-specification with each GET or PUT statement. The statements can also include options that specify the origin or destination of the data values or indicate where they appear in the stream relative to the preceding data values. Options for the

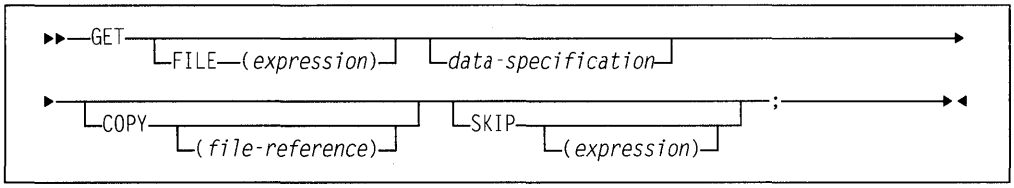
stream-data transmission statements are described in "Options of data transmission statements" on page 254.

GET statement

The GET statement is a STREAM input data transmission statement that can either:

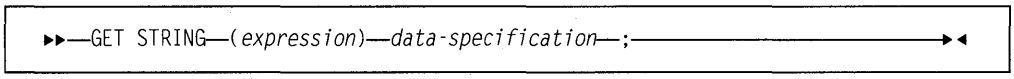
- Assign data values from a data set to one or more variables
- Assign data values from a string to one or more variables.

The syntax of the GET statement for a stream input file is:



The keywords can appear in any order. The data specification must appear unless the SKIP option is specified.

The syntax of the GET statement for transmission from a string is:



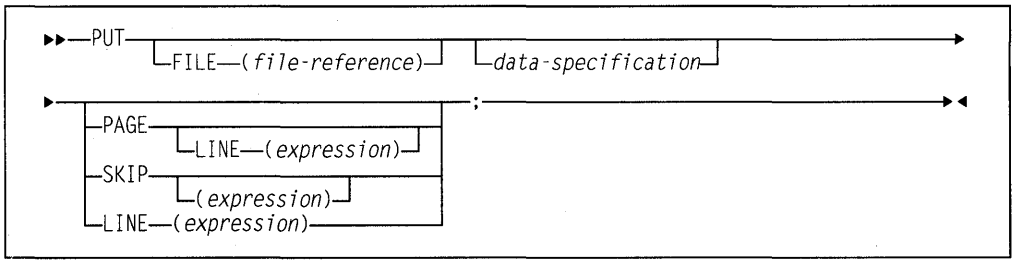
If FILE or STRING option is not specified FILE(SYSIN) is assumed and SYSIN is implicitly declared FILE STREAM INPUT EXTERNAL.

PUT statement

The PUT statement is a STREAM output data transmission statement that can:

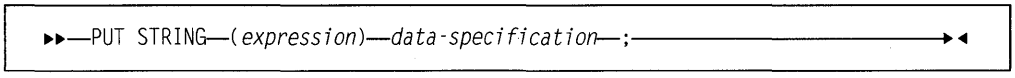
- Transmit values to a stream output file
- Assign values to a character variable.

The syntax of the PUT statement for a stream output file is:



The keywords can appear in any order. The data specification can be omitted only if one of the control options (PAGE, SKIP, or LINE) appears.

The syntax of the PUT statement for transmission to a character string is:



Options of data transmission statements

COPY option

The COPY option specifies that the source data stream will be written on the specified STREAM OUTPUT file without alteration. If no file reference is given, the default is the output file SYSPRINT. Each new record in the input stream starts a new record on the COPY file. For example:

```
get file(sysin) data(A,B,C) copy(DPL);
```

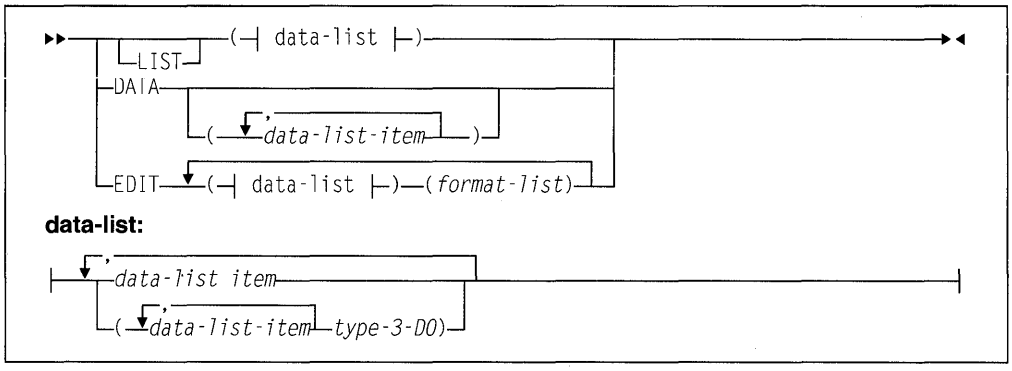
not only transmits the values assigned to A, B, and C in the input stream to the variables with these names, but also writes them exactly as they appear in the input stream, on the file DPL. Data values that are skipped on input, and not transmitted to internal variables, copy intact into the output stream.

If a condition is raised during the execution of a GET statement with a COPY option and an ON-unit is entered in which another GET statement is executed for the same file, and if control is returned from the ON-unit to the first GET statement, that statement executes as if no COPY option was specified. If, in the ON-unit, a PUT statement is executed for the file associated with the COPY option, the position of the data transmitted might not immediately follow the most recently-transmitted COPY data item.

If the COPY option file is not open in the current program, the file is implicitly opened in the program for stream output transmission.

Data specification options

Data specifications in GET and PUT statements specify the data to be transmitted. The syntax for a data specification is:



If a GET or PUT statement includes a data list that is not preceded by one of the keywords LIST, DATA, or EDIT, LIST is the default.

Important: In a statement without LIST, DATA, or EDIT preceding the data list, the data list must **immediately** follow the GET or PUT keyword. Any options required must be specified after the data list.

DATA Refer to "Data-directed data specification" on page 259.

EDIT Refer to "Edit-directed data specification" on page 263.

LIST Refer to "List-directed data specification" on page 267.

data-list item

On input, a data-list item for edit-directed and list-directed transmission can be one of the following: an element, array, or structure variable. For a data-directed data specification, a data-list item can be an element, array, or structure variable. None of the names in a data-directed data list can be subscripted or locator-qualified. However, qualified (that is, structure-member) or string-overlay-defined names are allowed.

On output, a data list item for edit-directed and list-directed data specifications can be an element expression, an array expression or a structure expression. For a data-directed data specification, a data-list item can be an element, array, or structure variable. It must not be locator-qualified. It can be qualified (that is, a member of a structure) or string-overlay-defined.

The data types of a data-list item can be any computational data. In this case, the name of the variable is transmitted, but not its value.

An array or structure variable in a data-list is equivalent to *n* items in the data list, where *n* is the number of element items in the array or structure. For edit-directed transmission, each element item is associated with a separate use of a data-format item.

data-list item type-3-DO

The syntax for the Type 3 DO specification is described under "DO statement" on page 173. Data list items with Type 3 DO specifications are not permitted in data-directed data lists.

When the last repetitive specification is completed, processing continues with the next data-list item.

Each repetitive specification must be enclosed in parentheses, as shown in the syntax diagram. If a data specification contains only a repetitive specification, two sets of outer parentheses are required, since the data list is enclosed in parentheses and the repetitive specification must have a separate set.

When repetitive specifications are nested, the rightmost DO is at the outer level of nesting. For example:

```
get list (((A(I,J)
     do I = 1 to 2)
     do J = 3 to 4));
```

There are three sets of parentheses, in addition to the set used to delimit the subscripts. The outermost set is the set required by the data specification. The next set is that required by the outer repetitive specification. The third set of parentheses is required by the inner repetitive specification.

This statement is equivalent in function to the following nested do-groups:

```
do J = 3 to 4;
   do I = 1 to 2;
     get list (A (I,J));
   end;
end;
```

It assigns values to the elements of the array A in the following order:

```
A(1,3), A(2,3), A(1,4), A(2,4)
```

format list
For a description of the format list, see "Edit-directed data specification" on page 263.

FILE option

The FILE option specifies the file upon which the operation takes place. It must be a STREAM file. For information on how to declare a file type data item, see "Files" on page 230.

If neither the FILE option nor the STRING option appears in a GET statement, the input file SYSIN is the default; if neither option appears in a PUT statement, the output file SYSPRINT is the default.

LINE option

The LINE option can be specified only for PRINT files. The LINE option defines a new current line for the data set. The expression is evaluated and converted to an integer value, *n*. The new current line is the *n*th line of the current page. If at least *n* lines have already been written on the current page or if *n* exceeds the limits set by the PAGESIZE option of the OPEN statement, the ENDPAGE condition is raised. If *n* is less than or equal to zero, a value of 1 is used. If *n* specifies the current line, ENDPAGE is raised except when the file is positioned on column 1, in which case the effect is the same as if a SKIP(0) option were specified.

The LINE option takes effect before the transmission of any values defined by the data specification (if any). If both the PAGE option and the LINE option appear in the same statement, the PAGE option is applied first. For example:

```
put file(List) data(P,Q,R) line(34) page;
```

prints the values of the variables P, Q, and R in data-directed format on a new page, commencing at line 34.

For the effect of the LINE option when specified in the first GET statement following the opening of the file, see "OPEN statement" on page 236.

For output to a terminal in interactive mode, the LINE option skips three lines.

PAGE option

The PAGE option can be specified only for PRINT files. It defines a new current page within the data set. If PAGE and LINE appear in the same PUT statement, the PAGE option is applied first. The PAGE option takes effect before the transmission of any values defined by the data specification (if any).

The page remains current until the execution of a PUT statement with the PAGE option, until a PAGE format item is encountered, or until the ENDPAGE condition is raised, resulting in the definition of a new page. A new current page implies line one.

For output to a terminal in interactive mode, the PAGE option skips three lines.

SKIP option

The SKIP option specifies a new current line (or record) within the data set. The expression is evaluated and converted to an integer value, *n*. The data set is positioned to the start of the *n*th line (record) relative to the current line (record). If *expression* is not specified, the default is SKIP(1).

The SKIP option takes effect before the transmission of values defined by the data specification (if any). For example:

```
put list(X,Y,Z) skip(3);
```

prints the values of the variables `X`, `Y`, and `Z` on the output file SYSPRINT commencing on the third line after the current line.

For non-PRINT files and input files, if the expression in the SKIP option is less than or equal to zero, a value of 1 is used. For PRINT files, if *n* is less than or equal to zero, the positioning is to the start of the current line.

For the effect of the SKIP option when specified in the first GET statement following the opening of the file, see "OPEN statement" on page 236.

If fewer than *n* lines remain on the current page when a SKIP(n) is issued, ENDPAGE is raised.

When printing at a terminal in conversational mode, SKIP(n) with *n* greater than 3 is equivalent to SKIP(3). No more than three lines can be skipped.

STRING option

The STRING option in GET and PUT statements transmits data between main storage locations rather than between the main and a data set. DBCS data items cannot be used with the STRING option.

The GET statement with the STRING option specifies that data values assigned to the data list items are obtained from the expression, after conversion to character string. Each GET operation using this option always begins at the leftmost character position of the string. If the number of characters in this string is less than the total number of characters specified by the data specification, the ERROR condition is raised.

The PUT statement with the STRING option specifies that values of the data-list items are to be assigned to the specified character variable or pseudovariable. The PUT operation begins assigning values at the leftmost character position of the string, after appropriate conversions are performed. Blanks and delimiters are inserted as in normal I/O operations. If the string is not long enough to accommodate the data, the ERROR condition is raised.

The NAME condition is not raised for a GET DATA statement with the STRING option. Instead, the ERROR condition is raised for situations that raise the NAME condition for a GET DATA statement with the FILE option.

The following restrictions apply to the STRING option:

- The COLUMN control format option cannot be used with the STRING option.
- No pseudovariables are allowed in the STRING option of a PUT statement.

The STRING option is most useful with edit-directed transmission. It allows data gathering or scattering operations performed with a single statement, and it allows stream-oriented processing of character strings that are transmitted by record-oriented statements.

For example:

```
read file (Inputr) into (Temp);
get string(Temp) edit (Code) (F(1));
If Code = 1 then
   get string (Temp) Edit (X,Y,Z)
      (X(1), 3 F(10,4));
```

The READ statement reads a record from the input file `Inputr`. The first GET statement uses the STRING option to extract the code from the first byte of the record and assigns it to `Code`. If the code is 1, the second GET statement uses the STRING option to assign the values in the record to `X`, `Y`, and `Z` The second GET statement specifies that the first character in the string `Temp` is ignored (the `X(1)` format item in the format list). This ignored character is the same one assigned to `Code` by the first GET statement.

An example of the STRING option in a PUT statement is:

```
put string (Record) edit
   (Name)    (X(1), A(12))
   (Pay#)    (X(10), A(7))
   (Hours*Rate) (X(10), P'$999V.99');

write file (Outprt) from (Record);
```

The PUT statement specifies, by the `X(1)` spacing format item, that the first character assigned to the character variable is to be a single blank, which is the ANS vertical carriage positioning character that specifies a single space before printing. Following that, the values of the variables `Name` and `Pay#` and of the expression `Hours*Rate` are assigned. The WRITE statement specifies that record transmission is used to write the record into the file `Outprt`.

The variable referenced in the STRING option should not be referenced by name or by alias in the data list. For example:

```
declare S char(8) init('YYMMDD');
put string (S) edit
    (substr (S, 3, 2), '/',
    substr (S, 5, 2), '/',
    substr (S, 1, 2))
    (A);
```

The value of `S` after the PUT statement is 'MM/bb/MM' and not 'MM/DD/YY' because `S` is blanked after the first data item is transmitted. The same effect is obtained if the data list contains a variable based or defined on the variable specified in the STRING option.

Transmission of data-list items

If a data-list item is of complex mode, the real part is transmitted before the imaginary part.

If a data-list item is an array expression, the elements of the array are transmitted in row-major order; that is, with the rightmost subscript of the array varying most frequently.

If a data-list item is a structure expression, the elements of the structure are transmitted in the order specified in the structure declaration.

For example, the statements

```
declare 1 A (10),
        2 B,
        2 C;
put file(X) list(A);
```

result in the output being ordered as follows:

```
A.B(1) A.C(1) A.B(2) A.C(2) A.B(3)
    A.C(3)...
```

If, however, the declaration is:

```
declare 1 A,
          2 B(10),
          2 C(10);
```

the same PUT statement results in the output ordered as follows:

```
A.B(1) A.B(2) A.B(3) ... A.B(10)
A.C(1) A.C(2) A.C(3) ... A.C(10)
```

If an input statement for list- or edit-directed transmission assigns a value to a variable in a data list, the assigned value is used if the variable appears in a later reference in the data list. For example:

```
get list (N,(X(I) do I=1 to N),J,K,);
     substr (Name, J,K));
```

When this statement is executed, values are transmitted and assigned in the following order:

1. A new value is assigned to `N`.
2. Elements are assigned to the array `X` as specified in the repetitive specification in the order `X(1),X(2),...X(N)`, with the new value of `N` specifying the number of assigned items.
3. A new value is assigned to `J`.
4. A new value is assigned to `K`

Data-directed data specification

For a description of the syntax of the DATA data specification, refer to "Data specification options" on page 254.

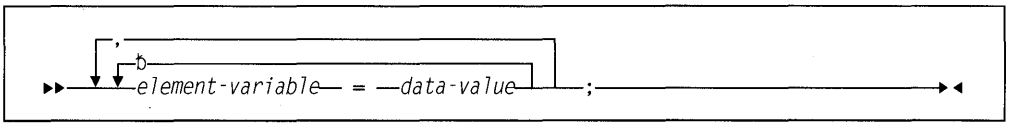
Names of structure elements in the data-list item need only have enough qualification to resolve any ambiguity. Full qualification is not required.

Omission of the data list results in a default data list that contains all computational variables that could be named in a data-directed statement.

On output, all items in the data list are transmitted. If two or more blocks containing the PUT statement each have declarations of items that have the same name, all the items are transmitted. The item in the innermost block appears first. Parameters, based variables, defined variables, and subscripted variables are not permitted in data-directed lists.

Syntax of data-directed data

The stream associated with data-directed data transmission is in the form of a list of element assignments. The element assignments that have optionally signed constants, like variable names and equal signs, are in character or graphic form. The syntax for the element assignments is:



On input, the element assignments can be separated by either a blank or a comma. Blanks can surround periods in qualified names, subscripts, subscript parentheses, and the assignment symbols. On output, the assignments are separated by a blank. For PRINT files, items are separated according to program tab settings.

Each data-value in the stream has one of the syntaxes described for list-directed transmission. For a description of list-directed transmission syntax, refer to "Syntax of list-directed data" on page 268.

The length of the data value in the stream is a function of the attributes declared for the variable and, because the name is also included, the length of the fully qualified subscripted name. The length for output items converted from coded arithmetic data, numeric character data, and bit-string data is the same as that for list-directed output data, and is governed by the rules for data conversion to character type as described in Chapter 5, "Data conversion."

Qualified names in the input stream must be fully qualified.

Interleaved subscripts cannot appear in qualified names in the stream. For example, assume that `Y` is declared as follows:

```
declare 1 Y(5,5),
          2 A(10),
            3 B,
            3 C,
            3 D;
```

An element name has to appear in the stream as follows:

```
Y.A.B(2,3,8)= 8.72
```

GET data-directed

For more information about the GET statement, see "GET statement" on page 253.

If a data list is used, each data-list item must be an element, array, or structure variable. Names cannot be subscripted, but qualified names are allowed in the data list. All names in the stream should appear in the data list; however, the order of the names need not be the same, and the data list can include names that do not appear in the stream.

If the data list contains a name that is not included in the stream, the value of the named variable remains unchanged.

If the stream contains an unrecognizable element-variable or a name that does not have a counterpart in the data list, the NAME condition is raised.

Transmission ends when a semicolon that is not enclosed in quotation marks or an end-of-file is reached. The recognition of the semicolon or end-of-file determines the number of element assignments that are actually transmitted by a particular statement, whether or not a data list is specified.

For example, consider the following data list, where `A`, `B`, `C`, and `D` are names of element variables:

```
Data (B, A, C, D)
```

This data list can be associated with the following input data stream:

```
A= 2.5, B= .0047, D= 125, Z= 'ABC';
```

Because `C` appears in the data list but not in the stream, its value remains unaltered. `Z`, which is not in the data list, raises the NAME condition.

If the data list includes the name of an array, subscripted references to that array can appear in the stream although subscripted names cannot appear in the data list. The entire array need not appear in the stream; only those elements that actually appear in the stream will be assigned. If a subscript is out of range, or is missing, the NAME condition is raised.

For example:

```
declare X (2,3);
```

Consider the following data list and input data stream:

Data Specification	Input Data Stream
`data (X)`	`X(1,1)= 7.95,`
	`X(1,2)= 8085,`
	`X(1,3)= 73;`

Although the data list has only the name of the array, the input stream can contain values for individual elements of the array. In this case, only three elements are assigned; the remainder of the array is unchanged.

If the data list includes the names of structures, minor structures, or structure elements, fully qualified names must appear in the stream, although full qualification is not required in the data list. For example:

```
dcl 1 In,
      2 Partno,
      2 Descrp,
      2 Price,
        3 Retail,
        3 Whsl;
```

If it is desired to read a value for `In.Price.Retail`, the input data stream must have the following form:

```
In.Price.Retail=1.23;
```

The data specification can be any of:

```
data(In)
data(Price)
data(In.Price)
data(Retail)
data(Price.Retail)
data(In.Retail)
data(In.Price.Retail)
```

PUT data-directed

For more information about the PUT statement, see "PUT statement" on page 253.

A data-list item can be an element, array, or structure variable, or a repetitive specification. Subscripted names are not allowed. The names appearing in the data list, together with their values, are transmitted in the form of a list of element assignments separated by blanks and terminated by a semicolon. For PRINT files, items are separated according to program tab settings; see "PRINT attribute" on page 270.

A semicolon is written into the stream after the last data item transmitted by each PUT statement.

Names are transmitted as all SBCS or all DBCS, regardless of how they are specified in the data list. If a name contains only SBCS characters, it is transmitted as all SBCS; otherwise, it is transmitted as DBCS. Each name in a qualified reference is handled independently. For example:

```
put data (.A.B.C.Skk);
```

would be transmitted as:

```
ABCSkk=value-of-variable
```

Note: In the previous example, `.A.B.C.Skk` is a scalar variable.

Data-directed output is not valid for subsequent data-directed input when the character-string value of a numeric character variable does not represent a valid optionally signed arithmetic constant, or a complex expression.

For character data, the contents of the character string are written out enclosed in quotation marks. Each quotation mark contained within the character string is represented by two successive quotation marks.

The following example shows data-directed transmission (both input and output):

```
declare (A(6), B(7)) fixed;
get file (X) data (B);
do I  = 1 to 6;
     A (I) = B (I+1) + B (I);
end;
put file (Y) data (A);
```

input stream:

```
B(1)=1, B(2)=2, B(3)=3,
B(4)=1, B(5)=2, B(6)=3, B(7)=4;
```

output stream:

```
A(1)= 3 A(2)= 5 A(3)= 4 A(4)= 3
A(5)= 5 A(6)= 7;
```

In the following example:

```
dcl 1 A,
      2 B FIXED,
      2 C,
        3 D FIXED;
A.B = 2;
A.D = 17;
put data (A);
```

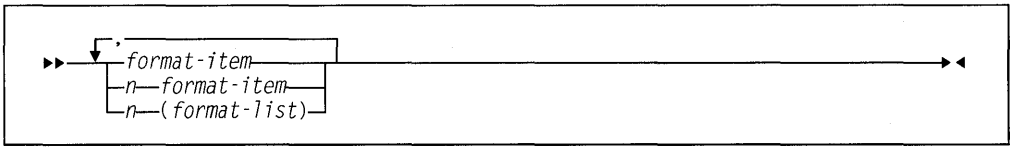
the data fields in the output stream are as follows:

```
A.B= 2 A.C.D= 17;
```

Edit-directed data specification

For information on the syntax of the EDIT data specification, refer to "Data specification options" on page 254.

The syntax for an edit-directed format list specification is:



n specifies an iteration factor, which is either an expression enclosed in parentheses or an integer. If it is the latter, a blank must separate the integer and the following format item.

The iteration factor specifies that the associated format item or format list is used *n* successive times. A zero or negative iteration factor specifies that the associated format item or format list is skipped and not used (the data-list item is associated with the next data-format item).

If an expression is used to represent the iteration factor, it is evaluated and converted to an integer, once for each set of iterations.

The associated format item or format list is that item or list of items immediately to the right of the iteration factor.

format item
specifies either a data-format item, a control-format item, or the remote format item. Syntax and detailed discussions of the format items appear in Chapter 13, "Edit-directed format items."

Data-format items
describe the character or graphic representation of a single data item. They are:

A character
B bit
C complex
E floating point
F fixed point
G graphic
P picture

Control-format items
specify the layout of the data set associated with a file. They are:

COLUMN
LINE
PAGE
SKIP
X

The remote-format item
specifies a label reference whose value is the label constant of a FORMAT statement located elsewhere. The FORMAT statement contains the remotely situated format items. The label reference item is:

R(label-reference)
where label is the label constant name of the FORMAT statement. For information on specifying the R-format item, see "R-format item" on page 282.

The first data-format item is associated with the first data-list item, the second data-format item with the second data-list item, and so on. If a format list contains fewer data-format items than there are items in the associated data list, the format list is reused. If there are excessive format items, they are ignored.

Suppose a format list contains five data-format items and its associated data list specifies ten items to be transmitted. The sixth item in the data list is associated with the first data-format item, and so forth. Suppose a format list contains ten data-format items and its associated data list specifies only five items. The sixth through the tenth format items are ignored.

If a control-format item is encountered, the control action is executed.

The PAGE and LINE control-format items can be used only with PRINT files and, consequently, can appear only in PUT statements. The SKIP, COLUMN, and X-format items apply to both input and output.

The PAGE, SKIP, and LINE format items have the same effect as the corresponding options of the PUT statement (and of the GET statement, in the case of SKIP), except that the format items take effect when they are encountered in the format list, while the options take effect before any data is transmitted.

The COLUMN format item cannot be used in a GET STRING or PUT STRING statement.

For the effects of control-format items when specified in the first GET or PUT statement following the opening of a file, see "OPEN statement" on page 236.

A value read into a variable can be used in a format item that is associated with another variable later in the data list.

```
get edit (M,String_A,I,String_B)(F(2),A(M),X(M),F(2),A(I));
```

In this example, the first two characters are assigned to `M` The value of `M` specifies the number of characters assigned to `String_A` and the number of characters being ignored before two characters are assigned to `I`, whose value is used to specify the number of characters assigned to `String_B`.

The value assigned to a variable during an input operation can be used in an expression in a format item that is associated with a later data item. An expression in a format item is evaluated and converted to an integer each time the format item is used.

The transmission is complete when the last data-list item has been processed. Subsequent format items, including control-format items, are ignored.

GET edit-directed

For more information about the GET statement, see "GET statement" on page 253.

Data in the stream is a continuous string of characters and graphics with no delimiters between successive values. The number of characters for each data value is specified by a format item in the format list. The characters are interpreted according to the associated format item. When the data list has been processed, execution of the GET statement stops and any remaining format items are not processed.

Each data-format item specifies the number of characters or graphics to be associated with the data-list item and how to interpret the data value. The data value is assigned to the associated data-list item, with any necessary conversion.

Fixed-point binary and floating-point binary data values must always be represented in the input stream with their values expressed in decimal digits. The F-, P-, and E-format items can then be used to access them, and the values are converted to binary representation upon assignment.

All blanks and quotation marks are treated as characters in the stream. Strings should not be enclosed in quotation marks. Quotation marks should not be doubled. The letter B should not be used to identify bit strings or G to identify graphic strings. If characters in the stream cannot be interpreted in the manner specified, the CONVERSION condition is raised.

Example:

```
get edit (Name, Data, Salary)(A(N), X(2), A(6), F(6,2));
```

This example specifies the following:

- The first N characters in the stream are treated as a character string and assigned to Name.
- The next two characters are skipped.
- The next six characters are assigned to Data in character format.
- The next six characters are considered an optionally signed decimal fixed-point constant and assigned to Salary.

PUT edit-directed

For more information about the PUT statement, see "PUT statement" on page 253.

The value of each data-list item is converted to the character or graphic representation specified by the associated data-format item and placed in the stream in a field whose width also is specified by the format item. When the data list has been processed, execution of the PUT statement stops and any remaining format items are not processed.

On output, binary items are converted to decimal values and the associated F- or E-format items must state the field width and point placement in terms of the converted decimal number. For the P-format these are specified by the picture specification.

On output, blanks are not inserted to separate data values in the output stream. String data is left-adjusted in the field to the width specified. Arithmetic data is right-adjusted. Because of the rules for conversion of arithmetic data to character type which can cause up to 3 leading blanks to be inserted (in addition to any blanks that replace leading zeros), generally there is at least 1 blank preceding an arithmetic item in the converted field. Leading blanks do not appear in the stream, however, unless the specified field width allows for them. Truncation, due to inadequate field-width specification, is on the left for arithmetic items, and on the right for string items. SIZE or STRINGSIZE is raised if truncation occurs.

Example 1

```
put edit('Inventory='||Inum,Invcode)(A,F(5));
```

This example specifies that the character string 'Inventory=' is concatenated with the value of Inum and placed in the stream in a field whose width is the length of the resultant string. Then the value of Invcode is converted to character, as described by the F-format item, and placed in the stream right-adjusted in a field with a width of five characters (leading characters can be blanks).

Example 2: The following example shows the use of the COLUMN, LINE, PAGE, and SKIP format items in combination with one another:

```
put edit ('Quarterly Statement')
    (page, line(2), A(19))(Acct#, Bought, Sold, Payment, Balance)
    (skip(3), A(6), column(14), F(7,2), column(30), F(7,2),
    column(45), F(7,2), column(60), F(7,2));
```

This PUT statement specifies the following:

1. The heading Quarterly Statement is written on line two of a new page in the output file SYSPRINT.

2. Two lines are skipped. The next line in the output is the third line following the heading, or the fifth line of the report.
3. The following values are written:

   Acct#, beginning at character position 1
   Bought, beginning at character position 14
   Sold, beginning at character position 30
   Payment, beginning at character position 45
   Balance at character position 60.

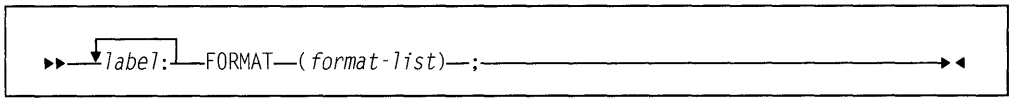
Example 3: In the following example, the value of Name is inserted in the stream as a character string left-adjusted in a field of N characters.

```
put edit (Name,Number,City) (A(N),A(N-4),A(10));
```

Number is left-adjusted in a field of N-4 characters; and City is left-adjusted in a field of 10 characters.

FORMAT statement

The FORMAT statement specifies a format list that can be used by edit-directed data transmission statements to control the format of the data being transmitted.



label This label constant is the same as the label-reference of the remote-format item, R, discussed in "R-format item" on page 282.

format list
is specified as described under "Edit-directed data specification" on page 263.

A GET or PUT EDIT statement can include an R-format item in its format-list option. That portion of the format list represented by the R-format item is supplied by the identified FORMAT statement.

A condition prefix associated with a FORMAT statement is invalid.

List-directed data specification

For information on the syntax of the LIST data specification, refer to "Data specification options" on page 254.

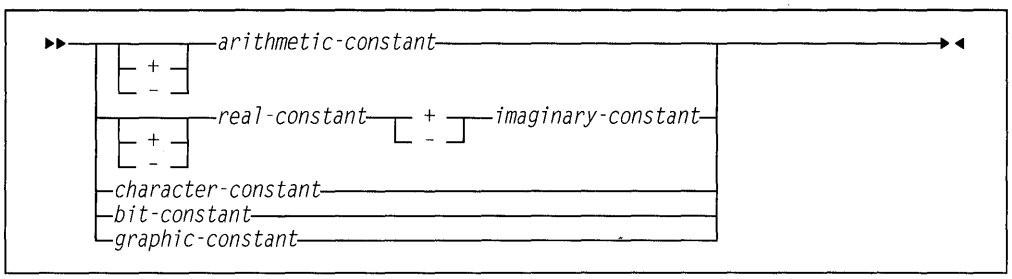
Examples of list-directed data specifications are:

```
list (Card_Rate, Dynamic_Flow)

list ((Thickness(Distance)
     do Distance = 1 to 1000))

list (P, Z, M, R)

list (A*B/C, (X+Y)**2)
```

The specification in the last example can be used only for output, since it contains expressions. These expressions are evaluated when the statement is executed, and the result is placed in the stream.

Syntax of list-directed data

Data values in the stream, either input or output, are character or graphic representations. The syntax for data values is:



String repetition factors are not allowed. A blank must not follow a sign preceding a real constant, and must not precede or follow the central positive (+) or negative (–) symbol in complex expressions.

The length of the data value in the stream is a function of the attributes of the data value, including precision and length. Detailed discussions of the conversion rules and their effect upon precision are listed in the descriptions of conversion to character type in Chapter 5, "Data conversion" on page 72.

GET list-directed

For information about the GET statement, see "GET statement" on page 253.

On input, data values in the stream must be separated either by a blank or by a comma. This separator can be surrounded by one or more blanks. A null field in the stream is indicated either by the first nonblank character in the data stream being a comma, or by two commas separated by an arbitrary number of blanks. A null field specifies that the value of the associated data-list item remains unchanged.

Transmission of the list of constants or complex expressions on input is terminated by expiration of the list or at the end-of-file. For transmission of constants, the file is positioned in the stream ready for the next GET statement.

If the items are separated by a comma, the first character scanned when the next GET statement is executed is the one immediately following the comma:

```
Xbb,bbbXX
   ↑
```

If the items are separated by blanks only, the first item scanned is the next nonblank character:

```
XbbbbXXX
    ↑
```

unless the end-of-record is encountered, in which case the file is positioned at the end of the record:

```
Xbb-bbXXX
   ↑
```

However, if the end-of-record immediately follows a nonblank character (other than a comma), and the following record begins with blanks, the file is positioned at the first nonblank character in the following record:

```
X-bbbXXX
     ↑
```

If the record does terminate with a comma, the next record is not read until the next GET statement requires it.

If the data is a character constant, the surrounding quotation marks are removed, and the enclosed characters are interpreted as a character string. A double quotation mark is treated as a single quotation mark.

If the data is a bit constant, the enclosing quotation marks and the trailing character B are removed, and the enclosed characters are interpreted as a bit string.

If the data is a hexadecimal constant (X, BX, B4, GX), the enclosing quotation marks and the suffix are removed, and the enclosed characters are interpreted as a hexadecimal representation of a character, bit, or graphic string.

If the data is a mixed constant, the enclosing quotation marks and the suffix M are removed, and the enclosed constant is interpreted as a character string.

If the data is a graphic constant, the enclosing quotation marks and the trailing character G are removed, and the enclosed graphics are interpreted as a graphic string.

If the data is an arithmetic constant or complex expression, it is interpreted as coded arithmetic data with the base, scale, mode, and precision implied by the constant or by the rules for expression evaluation.

PUT list-directed

For more information about the PUT statement, see "PUT statement" on page 253.

The values of the data-list items are converted to character representations (except for graphics) and transmitted to the data stream. A blank separates successive data values transmitted. For PRINT files, items are separated according to program tab settings (see "PRINT attribute" on page 270).

Arithmetic values are converted to character.

Binary data values are converted to decimal notation before being placed in the stream.

For numeric character values, the character value is transmitted.

Bit strings are converted to character strings. The character string is enclosed in quotation marks and followed by the letter B.

Character strings are written out as follows:

- If the file does not have the attribute PRINT, enclosing quotation marks are supplied, and contained single quotation marks or apostrophes are replaced by two quotation marks. The field width is the current length of the string plus the number of added quotation marks.
- If the file has the attribute PRINT, enclosing quotation marks are not supplied, and contained single quotation marks or apostrophes are unmodified. The field width is the current length of the string.

Mixed strings are enclosed in SBCS quotation marks and followed by the letter M. Contained SBCS quotes are replaced by two quotes.

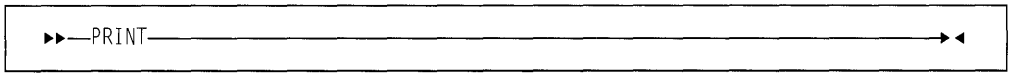
Graphic strings are written out as follows:

- If the file does not have the attribute PRINT, SBCS quotation marks, and the letter G are supplied. Because the enclosing quotation marks are SBCS, contained graphic quotation marks are represented by a single graphic quotation mark (unmodified).
- If the file has the attribute PRINT, graphic quotation marks are represented by a single graphic quotation mark (unmodified).

PRINT attribute

The PRINT attribute applies to files with the STREAM and OUTPUT attributes. It indicates that the file is intended to be printed; that is, the data associated with the file is to appear on printed pages, although it can first be written on some other medium.

The syntax for PRINT is:



When PRINT is specified, the first data byte of each record of a PRINT file is reserved for an American National Standard (ANS) printer control character. The control characters are inserted by PL/I.

Data values transmitted by list- and data-directed data transmission are automatically aligned on the left margin and on implementation-defined preset tab positions.

The layout of a PRINT file can be controlled by the use of the options and format items listed in Figure 44.

Figure 44 (Page 1 of 2). Options and format items for PRINT files

Statement	Statement Option	Edit directed format item	Effect
OPEN	LINESIZE(n)	–	Established line width
OPEN	PAGESIZE(n)	–	Establishes page length
PUT	PAGE	PAGE	Skip to new page
PUT	LINE(n)	LINE(n)	Skip to specified line

Figure 44 (Page 2 of 2). Options and format items for PRINT files

Statement	Statement Option	Edit directed format item	Effect
PUT	SKIP[(n)]	SKIP[(n)]	Skip specified number of lines
PUT	–	COLUMN(n)	Skip to specified character position in line
PUT	–	X(n)	Places blank characters in line to establish position.

LINESIZE and PAGESIZE establish the dimensions of the printed area of the page, excluding footings. The LINESIZE option specifies the maximum number of characters included in each printed line. If it is not specified for a PRINT file, a default value of 120 characters is used. There is no default for a non-PRINT file. The PAGESIZE option specifies the maximum number of lines in each printed page; if it is not specified, a default value of 60 lines is used. For example:

```
open file(Report) output stream print PAGESIZE(55) LINESIZE(110);
on endpage(Report) begin;
     put file(Report) skip list (Footing);
     Pageno = Pageno + 1;
     put file(Report) page list ('Page '||Pageno);
     put file(Report) skip (3);
end;
```

The OPEN statement opens the file Report as a PRINT file. The specification PAGESIZE(55) indicates that each page contains a maximum of 55 lines. An attempt to write on a page after 55 lines have already been written (or skipped) raises the ENDPAGE condition. The implicit action for the ENDPAGE condition is to skip to a new page, but you can establish your own action through use of the ON statement, as shown in the example.

LINESIZE(110) indicates that each line on the page can contain a maximum of 110 characters. An attempt to write a line greater than 110 characters places the excess characters on the next line.

When an attempt is made to write on line 56 (or to skip beyond line 55), the ENDPAGE condition is raised, and the begin block shown here is executed. The ENDPAGE condition is raised only once per page. Consequently, printing can be continued beyond the specified PAGESIZE after the ENDPAGE condition has been raised. This can be useful, for example, if you want to write a footing at the bottom of each page.

The first PUT statement specifies that a line is skipped, and the value of Footing, presumably a character string, is printed on line 57 (when ENDPAGE is raised, the current line is always PAGESIZE+1). The page number, `Pageno`, is incremented, the file `Report` is set to the next page, and the character constant `'Page'` is concatenated with the new page number and printed. The final PUT statement skips three lines, so that the next printing will be on line 4. Control returns from the begin block to the PUT statement that raised the ENDPAGE condition. However, any SKIP or LINE option specified in that statement has no further effect.

DBCS data in stream I/O

If DBCS data is used in list-directed or data-directed transmission, the GRAPHIC option of the ENVIRONMENT attribute must be specified for that file. It also must be specified if data-directed transmission uses DBCS names even though no DBCS data is present. DBCS continuation rules are applied and are the same rules as those described in "DBCS continuation rules" on page 21. For information on how graphics are handled for edit-directed transmission, see "Edit-directed data specification" on page 263.

Chapter 13. Edit-directed format items

Chapter 13. Edit-directed format items

This chapter describes each of the edit-directed format items that can appear in the format list of a GET, PUT, or FORMAT statement. (See also "Edit-directed data specification" on page 263.) The format items are described in alphabetic order.

A-format item

The character (or A) format item describes the representation of a character value.

field-width
specifies the number of character positions in the data stream that contain (or will contain) the string. It is an expression that is evaluated and converted to an integer value, which must be nonnegative, each time the format item is used.

On input, the specified number of characters is obtained from the data stream and assigned, with any necessary conversion, truncation, or padding, to the data-list item. The field width is always required on input and, if it is zero, a null string is obtained. If quotation marks appear in the stream, they are treated as characters in the string.

In the example:

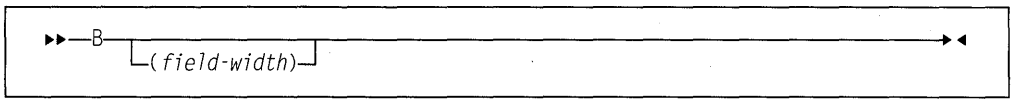
```
get file (Infile) edit (Item) (A(20));
```

The GET statement assigns the next 20 characters in `Infile` to `Item` The value is converted from its character representation specified by the format item `A(20)`, to the representation specified by the attributes declared for `Item`.

On output, the data-list item is converted, if necessary, to a character string and is truncated or extended with blanks on the right to the specified field-width before being placed into the data stream. If the field-width is zero, no characters are placed into the data stream. Enclosing quotation marks are never inserted, nor are contained quotation marks doubled. If the field width is not specified, the default is equal to the character-string length of the data-list item (after conversion, if necessary, according to the rules given in Chapter 5, "Data conversion").

B-format item

The bit (or B) format item describes the character representation of a bit value. Each bit is represented by the character zero or one.



field-width
specifies the number of data-stream character positions that contain (or will contain) the bit string. It is an expression that is evaluated and converted to an integer value, which must be nonnegative, each time the format item is used.

On input, the character representation of the bit string can occur anywhere within the specified field. Blanks, which may appear before and after the bit string in the field, are ignored. Any necessary conversion occurs when the bit string is assigned to the data-list item. The field width is always required on input, and if it is zero, a null string is obtained. Any character other than 0 or 1 in the string, including embedded blanks, quotation marks, or the letter B, raises the CONVERSION condition.

On output, the character representation of the bit string is left-adjusted in the specified field, and necessary truncation or extension with blanks occurs on the right. Any necessary conversion to bit-string is performed. No quotation marks are inserted, nor is the identifying letter B. If the field width is zero, no characters are placed into the data stream. If the field width is not specified, the default is equal to the bit-string length of the data-list item (after conversion, if necessary, according to the rules given in Chapter 5, "Data conversion").

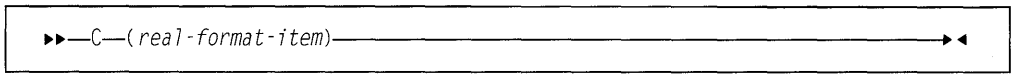
In the example:

```
declare Mask bit(25);
put file(Maskfle) edit (Mask) (B);
```

The PUT statement writes the value of `Mask` in `Maskfle` as a string of 25 characters consisting of zeros and ones.

C-format item

The complex (or C) format item describes the character representation of a complex data value. You use one real-format-item to describe both the real and imaginary parts of the complex data value in the data stream.



real-format-item
specified by one of the F-, E-, or P-format items. The P-format item must describe numeric character data.

On input, the letter I in the input raises the CONVERSION condition.

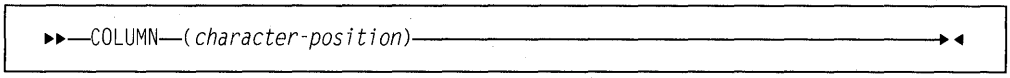
On output, the letter I is never appended to the imaginary part. If the second real format item (or the first, if only one appears) is an F or E item, the sign is transmitted only if the value of the imaginary part is less than zero. If the real format item is a P item, the sign is transmitted only if the S or - or + picture character is specified.

If you require an I to be appended, it must be specified as a separate data item in the data list, immediately following the variable that specifies the complex item.

The I, then, must have a corresponding format item (either A or P). If a second real format item is specified, it is ignored.

COLUMN Format item

The COLUMN format item positions the file to a specified character position within the current or following line.



character-position
specifies an expression which is evaluated and converted to an integer value, which must be nonnegative, each time the format item is used.

The file is positioned to the specified character position in the current line, provided it has not already passed this position. If the file is already positioned after the specified character position, the current line is completed and a new line is started; the format item is then applied to the following line.

Then, if the specified character position lies beyond the rightmost character position of the current line, or if the value of the expression for the character position is less than one, the default character position is one.

The rightmost character position is determined as follows:

- For output files, it is determined by the line size.
- For input files, it is determined using the length of the current logical record to determine the line size and, hence, the rightmost character position.

COLUMN must not be used in a GET STRING or PUT STRING statement.

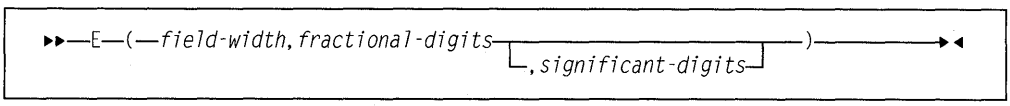
COLUMN cannot be used with input or output lines that contain graphics.

On input, intervening character positions are ignored.

On output, intervening character positions are filled with blanks.

E-format item

The floating-point (or E) format item describes the character representation of a real floating-point decimal arithmetic data value.



field-width
specifies the total number of characters in the field. It is evaluated and converted to an integer value *w* each time the format item is used.

fractional-digits
specifies the number of digits in the mantissa that follow the decimal point. It is evaluated and converted to an integer value *d* each time the format item is used.

significant-digits
specifies the number of digits that must appear in the mantissa. It is evaluated and converted to an integer value *s* each time the format item is used.

The following must be true:

```
w >= s = d+1 or w = 0
```

and, when w ¬= 0

```
s > 0, d >=0
```

Note: The maximum allowed value for d is 16 for PL/I compiler for OS/2. This can limit the portability of your application.

On input, either the data value in the data stream is an optionally signed real decimal floating-point or fixed-point constant located anywhere within the specified field or the CONVERSION condition is raised. (For convenience, the **E** preceding a signed exponent can be omitted.)

The field width includes leading and trailing blanks, the exponent position, the positions for the optional plus or minus signs, the position for the optional letter **E**, and the position for the optional decimal point in the mantissa.

The data value can appear anywhere within the specified field; blanks can appear before and after the data value in the field and are ignored. If the entire field is blank, the CONVERSION condition is raised. When no decimal point appears, *fractional-digits* specifies the number of character positions in the mantissa to the right of the assumed decimal point. If a decimal point does appear in the number, it overrides the specification of *fractional-digits.*

If *field-width* is 0, there is no assignment to the data-list item.

The statement:

```
get file(A) edit (Cost) (E(10,6));
```

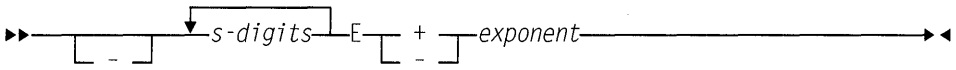
obtains the next 10 characters from `A` and interprets them as a floating-point decimal number. A decimal point is assumed before the rightmost 6 digits of the mantissa. The value of the number is converted to the attributes of `COST` and assigned to this variable.

On output, the data-list item is converted to floating-point and rounded if necessary. The rounding of data is as follows: if truncation causes a digit to be lost from the right, and this digit is greater than or equal to 5, 1 is added to the digit to the left of the truncated digit. This addition might cause adjustment of the exponent.

The character string written in the stream for output has one of the following syntaxes:

Note: Blanks are not permitted between the elements of the character strings.

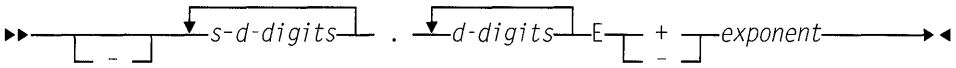
- For d=0



  w must be >=s+6 for positive values, or >=s+7 for negative values.

  When the value is nonzero, the exponent is adjusted so that the leading digit of the mantissa is nonzero. When the value is zero, zero suppression is applied to all digit positions (except the rightmost) of the mantissa.

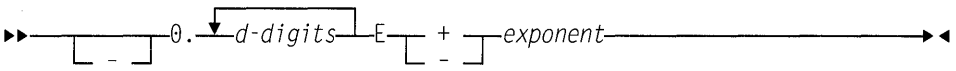
- For 0<d<s



  w must be >=s+7 for positive values, or >=s+8 for negative values.

  When the value is nonzero, the exponent is adjusted so that the leading digit of the mantissa is nonzero. When the value is zero, zero suppression is applied to all digit positions (except the first) to the left of the decimal point. All other digit positions contain zero.

- For d=s



  w must be >=d+8 for positive values, or >=d+9 for negative values.

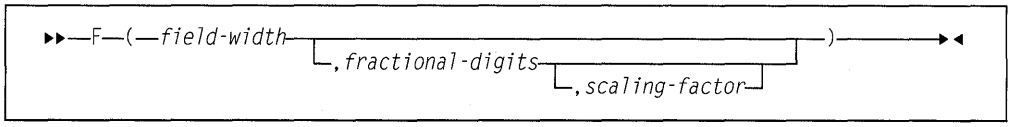
  When the value is nonzero, the exponent is adjusted so that the first fractional digit is nonzero. When the value is zero, each digit position contains zero.

The exponent is a 4-digit integer, which can be 4 zeros.

If the field width is such that significant digits or the sign are lost, the SIZE condition is raised. If the character string does not fill the specified field on output, the character string is right-adjusted and extended on the left with blanks.

F-format item

The fixed-point (or F) format item describes the character representation of a real fixed-point decimal arithmetic value.



field-width
specifies the total number of characters in the field. It is evaluated and converted to an integer value *w* each time the format item is used. The converted value must be nonnegative.

fractional-digits
specifies the number of digits in the mantissa that follow the decimal point. It is evaluated and converted to an integer value *d* each time the format item is used. The converted value must be nonnegative. If *fractional-digits* is not specified, the default value is 0.

scaling-factor
specifies the number of digits that must appear in the mantissa. It is evaluated and converted to an integer value *p* each time the format item is used.

On input, either the data value in the data stream is an optionally signed real decimal fixed-point constant located anywhere within the specified field or the CONVERSION condition is raised. Blanks can appear before and after the data value in the field and are ignored. If the entire field is blank, it is interpreted as zero.

If no *scaling-factor* is specified and no decimal point appears in the field, the expression for *fractional-digits* specifies the number of digits in the data value to the right of the assumed decimal point. If a decimal point does appear in the data value, it overrides the expression for *fractional-digits*.

If a *scaling-factor* is specified, it effectively multiplies the data value in the data stream by 10 raised to the integer value (*p*) of the *scaling-factor* Thus, if *p* is positive, the data value is treated as though the decimal point appeared *p* places to the right of its given position. If *p* is negative, the data value is treated as though the decimal point appeared *p* places to the left of its given position. The given position of the decimal point is that indicated either by an actual point, if it appears, or by the expression for *fractional-digits*, in the absence of an actual point.

If the *field-width* is 0, there is no assignment to the data-list item.

On output, the data-list item is converted, if necessary, to fixed-point. Floating point data converts to FIXED DECIMAL (N,q) where *q* is the *fractional-digits* specified. The data value in the stream is the character representation of a real decimal fixed-point number, rounded if necessary, and right-adjusted in the specified field.

The conversion from decimal fixed-point type to character type is performed according to the normal rules for conversion. Extra characters may appear as blanks preceding the number in the converted string. And, since leading zeros are converted to blanks (except for a 0 immediately to the left of the point), additional blanks may precede the number. If a decimal point or a minus sign appears, either will cause one leading blank to be replaced.

If only the *field-width* is specified, only the integer portion of the number is written; no decimal point appears.

If both the *field-width* and *fractional-digits* are specified, both the integer and fractional portions of the number are written. If the value (*d*) of *fractional-digits* is greater than 0, a decimal point is inserted before the rightmost *d* digits. Trailing zeros are supplied when *fractional-digits* is less than *d* (the value *d* must be less

than *field-width*). If the absolute value of the item is less than 1, a 0 precedes the decimal point. Suppression of leading zeros is applied to all digit positions (except the first) to the left of the decimal point.

The rounding of the data value is as follows: if truncation causes a digit to be lost from the right, and this digit is greater than or equal to 5, 1 is added to the digit to the left of the truncated digit.

On output, if the data-list item is less than 0, a minus sign is prefixed to the character representation; if it is greater than or equal to 0, no sign appears. Therefore, for negative values, the *field-width* might need to include provision for the sign, a decimal point, and a 0 before the point.

If the *field-width* is such that any character is lost, the SIZE condition is raised.

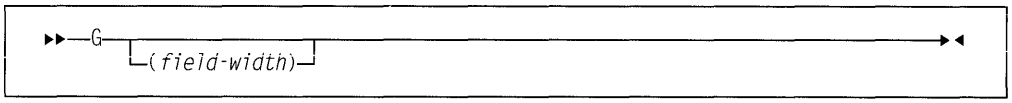
In the example:

```
declare Total fixed(4,2);
put edit (Total) (F(6,2));
```

The PUT statement specifies that the value of Total is converted to the character representation of a fixed-point number and written into the output file SYSPRINT. A decimal point is inserted before the last two numeric characters, and the number is right-adjusted in a field of six characters. Leading zeros are changed to blanks (except for a zero immediately to the left of the point), and, if necessary, a minus sign is placed to the left of the first numeric character.

G-format item

For the compiler, the graphic (or G) format item describes the representation of a graphic string.



field-width

specifies the number of 2-byte positions in the data stream that contain (or will contain) the graphic string. It is an expression that is evaluated and converted to an integer value, which must be nonnegative, each time the format item is used. End-of-line must not occur between the 2 bytes of a graphic.

On input, the specified number of graphics is obtained from the data stream and assigned, with any necessary truncation or padding, to the data-list item. The *field-width* is always required on input, and if it is zero, a null string is obtained.

On output, the data-list item is truncated or extended (with the padding graphic) on the right to the specified *field-width* before being placed into the data stream. No enclosing quotation marks are inserted, nor is the identifying suffix, *G*, inserted. If the *field-width* is zero, no graphics are placed into the data stream. If the *field-width* is not specified, it defaults to be equal to the graphic-string length of the data-list item.

In the following example, if OUT does not have the GRAPHIC option, six bytes are transmitted.

```
declare A graphic(3);
put file(Out) edit (A) (G(3));
```

L-format item

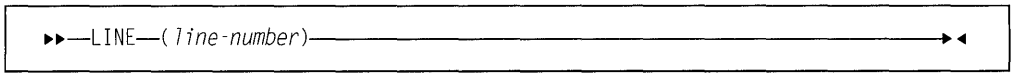
On input, L indicates that all data up to the end of the line is to be assigned to the data item.

On output, L indicates that the data item, padded on the right with blanks, if necessary, is to fill the remainder of the output line.

The syntax for the L format is:

LINE format item

The LINE format item specifies the line on the current page of a PRINT file upon which the next data-list item will be printed, or it raises the ENDPAGE condition.



line-number
can be represented by an expression, which is evaluated and converted to an integer value, which must be nonnegative, each time the format item is used.

Blank lines are inserted, if necessary.

If the specified *line-number* is less than or equal to the current line number, or if the specified line is beyond the limits set by the PAGESIZE option of the OPEN statement (or by default), the ENDPAGE condition is raised. An exception is that if the specified *line-number* is equal to the current line number, and the column 1 character has not yet been transmitted, the effect is as for a SKIP(0) item, that is, a carriage return with no line spacing.

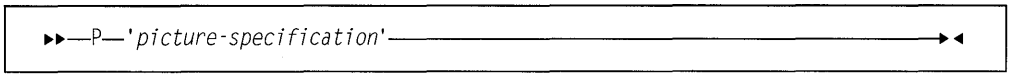
If *line-number* is zero, it defaults to one (1).

P-format item

The picture (or P) format item describes the character representation of real numeric character values and of character values.

The picture specification of the P-format item, on input, describes the form of the data item expected in the data stream and, in the case of a numeric character specification, how the item's arithmetic value is to be interpreted. If the indicated character does not appear in the stream, the CONVERSION condition is raised.

On output, the value of the associated element in the data list is converted to the form specified by the picture specification before it is written into the data stream.



picture-specification
is discussed in detail in Chapter 14, "Picture specification characters."

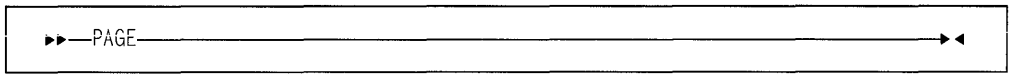
For example:

```
get edit (Name, Total) (P'AAAAA',P'9999');
```

When this statement is executed, the input file SYSIN is the default. The next five characters input from SYSIN must be alphabetic or blank and they are assigned to `Name` The next four characters must be digits and they are assigned to `Total`.

PAGE format item

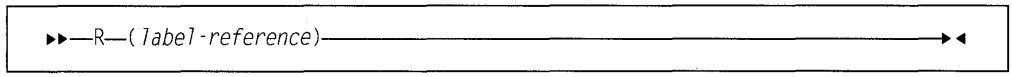
The PAGE format item specifies that a new page is established. It can be used only with PRINT files.



Starting a new page positions the file to the first line of the next page.

R-format item

The remote (or R) format item specifies that the format list in a FORMAT statement is to be used (as described under "FORMAT statement" on page 267).



label-reference
has as its value the label constant of a FORMAT statement.

The R-format item and the specified FORMAT statement must be internal to the same block, and they must be in the same invocation of that block.

A remote FORMAT statement cannot contain an R-format item that references itself as a label reference, nor can it reference another remote FORMAT statement that will lead to the referencing of the original FORMAT statement.

Conditions enabled for the GET or PUT statement must also be enabled for the remote FORMAT statement(s) that are referred to.

If the GET or PUT statement is the single statement of an ON-unit, that statement is a block, and it cannot contain a remote format item.

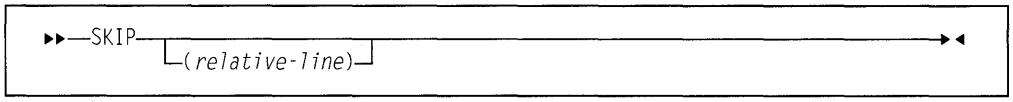
Example

```
declare Switch label;
get file(In) list(Code);
if Code = 1 then
  Switch = L1;
else
  Switch = L2;
get file(In) edit (W,X,Y,Z)
    (R(Switch));
L1:  format (4 F(8,3));
L2:  format (4 E(12,6));
```

`Switch` has been declared a label variable. The second GET statement can be made to operate with either of the two FORMAT statements.

SKIP format item

The SKIP format item specifies that a new line is to be defined as the current line.



relative-line
specifies an expression, which is evaluated and converted to an integer value, *n*, each time the format item is used. The converted value must be nonnegative and less than 32,768. It must be greater than zero for non-PRINT files. If it is zero, or if it is omitted, the default is 1.

The new line is the *n*th line after the present line.

If *n* is greater than one, one or more lines are ignored on input; on output, one or more blank lines are inserted.

The value *n* can be zero for PRINT files only, in which case the positioning is at the start of the current line. Characters previously written can be overprinted.

For PRINT files, if the specified *relative-line* is beyond the limit set by the PAGESIZE option of the OPEN statement (or the default), the ENDPAGE condition is raised.

If the SKIP format item is the first item to be executed after a file has been opened, output commences on the *n*th line of the first page. If *n* is zero or 1, it commences on the first line of the first page.

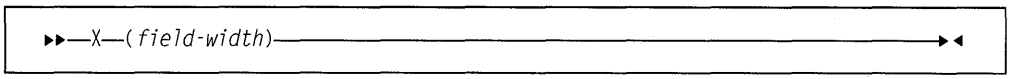
For example:

```
get file(In) edit(Man,Overtime)
     (skip(1), A(6), COL(60), F(4,2));
```

This statement positions the data set associated with file In to a new line. The first 6 characters on the line are assigned to Man, and the 4 characters beginning at character position 60 are assigned to Overtime.

X-format item

The spacing (or X) format item specifies the relative spacing of data values in the data stream.



field-width

specifies an expression that is evaluated and converted to an integer value, which must be nonnegative, each time the format item is used. The integer value specifies the number of characters before the next field of the data stream, relative to the current position in the stream.

On input, the specified number of characters are spaced over in the data stream and not transmitted to the program.

For example:

```
get edit (Number, Rebate)
       (A(5), X(5), A(5));
```

The next 15 characters from the input file, SYSIN, are treated as follows: the first five characters are assigned to Number, the next five characters are ignored, and the remaining five characters are assigned to Rebate.

On output, the specified number of blank characters are inserted into the stream.

In the example:

```
put file(Out) edit (Part, Count) (A(4), X(2), F(5));
```

Four characters that represent the value of Part, then two blank characters, and finally five characters that represent the fixed-point value of Count, are placed in the file named Out.

Chapter 14. Picture specification characters

Chapter 14. Picture specification characters

A picture specification consists of a sequence of picture characters enclosed in single or double quotation marks. The characters describe the contents of each position of the character or numeric character data item, and the contents of the output. The specification can be made in two ways:

- As part of the PICTURE attribute in a declaration
- As part of the P-format item (described in "P-format item" on page 282) for edit-directed input and output.

A picture specification describes either a character data item or a numeric character data item. The presence of an A or X picture character defines a picture specification as a character picture specification; otherwise, it is a numeric character picture specification.

A *character pictured item* can consist of alphabetic characters, decimal digits, blanks, currency and punctuation characters.

A *numeric character pictured item* can consist only of decimal digits, an optional decimal point, an optional letter E, and, optionally, one or two plus or minus signs. Other characters generally associated with arithmetic data, such as currency symbols, can also be specified, but they are not a part of the arithmetic value of the numeric character variable, although the characters are stored with the digits and are part of the character value of the variable.

Figures in this section illustrate how different picture specifications affect the representation of values when assigned to a pictured variable or when printed using the P-format item. Each figure shows the original value of the data, the attributes of the variable from which it is assigned (or written), the picture specification, and the character value of the numeric character or pictured character variable.

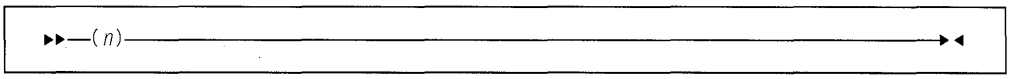
The concepts of the two types of picture specifications are described separately below.

Picture repetition factor

Repeating picture characters

A picture repetition factor specifies the number of repetitions of the next picture character in the specification.

The syntax is:



n is an integer. No blanks are allowed within the parentheses. If n is 0, the picture character is ignored.

For example, the following picture specifications result in the same description:

```
'999V99'
'(3)9V(2)9'
```

Picture characters for character data

A character picture specification describes a nonvarying character data item. You can specify that any position in the data item can contain only characters from certain subsets of the complete set of available characters. The data can consist of alphabetic characters, decimal digits, blanks,

The only valid characters in a character picture specification are X, A, and 9. Each of these specifies the presence of one character position in the character value, which can contain the following:

X Any character of the 256 possible bit combinations represented by the 8-bit byte.

A Any alphabetic or extralingual (#, @, $) character, or blank.

9 Any digit, or blank. (Note that the 9 picture specification character allows blanks only for character data.)

When a character value is assigned, or transferred, to a picture character data item, the particular character in each position is validated according to the corresponding picture specification character. If the character data does not match the specification for that position, the CONVERSION condition is raised for the invalid character. (However, if you change the value by record-oriented transmission or by using an alias, there is no checking.) For example:

```
declare Part# picture 'AAA99X';
put edit (Part#) (P'AAA99X');
```

The following values are valid for `Part#`:

```
'ABC12M'
'bbb09/'
'XYZb13'
```

The following values are not valid for `Part#` (the invalid characters are underscored):

```
'AB123M'
'ABC1/2'
'Mb#A5;'
```

Figure 45 shows examples of character picture specifications.

Figure 45. Character picture specification examples

Source Attributes	Source Data (in constant form)	Picture Specification	Character Value
CHARACTER(5)	'9B/2L'	XXXXX	9B/2L
CHARACTER(5)	'9B/2L'	XXX	9B/
CHARACTER(5)	'9B/2L'	XXXXXXX	9B/2Lbb
CHARACTER(5)	'ABCDE'	AAAAA	ABCDE
CHARACTER(5)	'ABCDE'	AAAAAA	ABCDEb
CHARACTER(5)	'ABCDE'	AAA	ABC
CHARACTER(5)	'12/34'	99X99	12/34
CHARACTER(5)	'L26.7'	A99X9	L26.7

Picture characters for numeric character data

Numeric character data represents numeric values. The picture specification cannot contain the character data picture characters X or A. The picture characters for numeric character data can also specify editing of the data.

A numeric character variable can have two values, depending upon how the variable is used. The types of values are as follows:

Arithmetic

The arithmetic value is the value expressed by the decimal digits of the data item, the assumed location of a decimal point, possibly a sign, and an optionally-signed exponent or scaling factor. The arithmetic value of a numeric character variable is used:

- Whenever the variable appears in an expression that results in a coded arithmetic value or bit value (this includes expressions with the ¬, &, |, and comparison operators; even comparison with a character string will use the arithmetic value of a numeric character variable)
- Whenever the variable is assigned to a coded arithmetic, numeric character, or bit variable
- When used with the C, E, F, B, and P (numeric) format items in edit-directed I/O.

The arithmetic value of the numeric character variable is converted to internal coded arithmetic representation.

Character value

The character value is the value expressed by the decimal digits of the data item, as well as all of the editing and insertion characters appearing in the picture specification. The character value does not, however, include the assumed location of a decimal point, as specified by the picture characters V, K, or F. The character value of a numeric character variable is used:

- Whenever the variable appears in a character expression
- In an assignment to a character variable
- Whenever the data is printed using list-directed or data-directed output
- Whenever a reference is made to a character variable that is defined or based on the numeric character variable
- Whenever the variable is printed using edit-directed output with the A or P (character) format items.

No data conversion is necessary.

Numeric character data can contain only decimal digits, an optional decimal point, an optional letter E, and one or two plus or minus signs. Other characters generally associated with arithmetic data, such as currency symbols, can also be specified, but they are not a part of the arithmetic value of the numeric character variable, although the characters are stored with the digits and are part of the character value of the variable.

A numeric character specification consists of one or more fields, each field describing a fixed-point number. A floating-point specification has two fields—one for the mantissa and one for the exponent. The first field can be divided into subfields by inserting a V picture specification character. The data preceding the V (if any) and that following it (if any) are subfields of the specification.

A requirement of the picture specification for numeric character data is that each field must contain at least one picture character that specifies a digit position. This picture character, however, need not be the digit character 9. Other picture characters, such as the zero suppression characters (Z or *), also specify digit positions.

Note: All characters except K, V, and F specify the occurrence of a character in the character representation.

The picture characters for numeric character specifications are discussed in the following sections:

- "Digits and decimal points" on page 290 describes data specified with the picture characters 9 and V.
- "Zero suppression" on page 291 describes picture data specified with the picture characters Z and asterisk (*).
- "Insertion characters" on page 292 discusses the use of the insertion characters (point, comma, slash, and B).
- "Insertion and decimal point characters" on page 293 describes the use of the decimal point and insertion characters with the V picture character.

- "Defining currency symbols" on page 294 describes how to define your own character(s) as a currency symbol, and "Signs and currency symbols" on page 296 describes the use of signs and currency symbols.
- "Credit, debit, and zero replacement characters." on page 298 discusses the picture characters CR, DB, and Y used for credit, debit, and zero replacement functions.
- "Exponent characters" on page 298 discusses the picture characters K and E used for exponents.
- "Scaling factor" on page 299 describes the picture character F used for scaling factors.
- "Picture repetition factor" on page 286 describes the picture repetition character.

Digits and decimal points

The picture characters 9 and V are used in numeric character specifications that represent fixed-point decimal values.

9 specifies that the associated position in the data item contains a decimal digit. (Note that the 9 picture specification character for numeric character data is different from the specification for character data because the corresponding character cannot be a blank for character data.)

A string of n 9 picture characters specifies that the item is a nonvarying character-string of length n, each of which is a digit (0 through 9). For example:

```
dcl digit picture'9',
    Count picture'999',
    XYZ picture '(10)9';
```

An example of use is:

```
dcl 1 Record,
      2 Data char(72),
      2 Identification char(3),
      2 Sequence pic'99999';
dcl Count fixed dec(5);
:
Count=Count+1;
Sequence=Count;
write file(Output) from(Record);
```

V specifies that a decimal point is assumed at this position in the associated data item. However, it does not specify that an actual decimal point or decimal comma is inserted. The integer value and fractional value of the assigned value, after modification by the optional scaling factor F(±x), are aligned on the V character. Therefore, an assigned value can be truncated or extended with zero digits at either end. (If significant digits are truncated on the left, the result is undefined and the SIZE condition is raised if enabled).

If no V character appears in the picture specification of a fixed-point decimal value (or in the first field of a picture specification of a floating-point decimal value), a V is assumed at the right end of the field specification. This can cause the assigned value to be truncated, if necessary, to an integer.

The V character cannot appear more than once in a picture specification.

For example:

```
dcl Value picture 'Z9V999';
Value = 12.345;
dcl Cvalue char(5);
Cvalue = Value;
```

`Cvalue`, after assignment of `Value`, contains `'12345'`.

Figure 46 shows examples of digit and decimal point characters.

Figure 46. Examples of digit and decimal point characters

Source Attributes	Source Data (in constant form)	Picture Specification	Character Value
FIXED(5)	12345	99999	12345
FIXED(5)	12345	99999V	12345
FIXED(5)	12345	999V99	undefined
FIXED(5)	12345	V99999	undefined
FIXED(7)	1234567	99999	undefined
FIXED(3)	123	99999	00123
FIXED(5,2)	123.45	999V99	12345
FIXED(7,2)	12345.67	9V9	undefined
FIXED(5,2)	123.45	99999	00123

Note: When the character value is undefined, the SIZE condition is raised.

Zero suppression

The picture characters Z and asterisk (*). specify conditional digit positions in the character value and can cause leading zeros to be replaced by asterisks or blanks. Leading zeros are those that occur in the leftmost digit positions of fixed-point numbers or in the leftmost digit positions of the two parts of floating-point numbers, that are to the left of the assumed position of a decimal point, and that are not preceded by any of the digits 1 through 9. The leftmost nonzero digit in a number and all digits, zeros or not, to the right of it represent significant digits.

Z specifies a conditional digit position and causes a leading zero in the associated data position to be replaced by a blank. Otherwise, the digit in the position is unchanged. The picture character Z cannot appear in the same field as the picture character * or a drifting character, nor can it appear to the right of any of the picture characters in a field.

***** specifies a conditional digit position. It is used the way the picture character Z is used, except that leading zeros are replaced by asterisks. The picture character asterisk cannot appear in the same field as the picture character Z or a drifting character, nor can it appear to the right of any of the picture characters in a field.

Figure 47 shows examples of zero suppression characters.

Figure 47. Examples of zero suppression characters

Source Attributes	Source Data (in constant form)	Picture Specification	Character Value
FIXED(5)	12345	ZZZ99	12345
FIXED(5)	00100	ZZZ99	␣␣100
FIXED(5)	00100	ZZZZZ	␣␣100
FIXED(5)	00000	ZZZZZ	␣␣␣␣␣
FIXED(5,2)	123.45	ZZZ99	␣␣123
FIXED(5,2)	001.23	ZZZV99	␣␣123
FIXED(5)	12345	ZZZV99	undefined
FIXED(5,2)	000.08	ZZZVZZ	␣␣␣08
FIXED(5,2)	000.00	ZZZVZZ	␣␣␣␣␣
FIXED(5)	00100	*****	**100
FIXED(5)	00000	*****	*****
FIXED(5,2)	000.01	***V**	***01
FIXED(5,2)	95	$**9.99	$**0.95
FIXED(5,2)	12350	$**9.99	$123.50

Note: When the character value is undefined, the SIZE condition is raised.

If one of the picture characters Z or asterisk appears to the right of the picture character V, all fractional digit positions in the specification, as well as all integer digit positions, must use the Z or asterisk picture character, respectively. When all digit positions to the right of the picture character V contain zero suppression picture characters, fractional zeros of the value are suppressed only if all positions in the fractional part contain zeros and all integer positions have been suppressed. The character value of the data item will then consist of blanks or asterisks. No digits in the fractional part are replaced by blanks or asterisks if the fractional part contains any significant digit.

Insertion characters

The picture characters comma (,), point (.), slash (/), and blank (B) cause the specified character to be inserted into the associated position of the numeric character data. They do not indicate digit or character positions, but are inserted between digits or characters. Each does, however, actually represent a character position in the character value, whether or not the character is suppressed. The comma, point, and slash are conditional insertion characters and can be suppressed within a sequence of zero suppression characters. The blank is an unconditional insertion character, It always specifies that a blank appears in the associated position.

Insertion characters are applicable only to the character value. They specify nothing about the arithmetic value of the data item. They never cause decimal point or decimal comma alignment in the picture specifications of a fixed-point decimal number and are not a part of the arithmetic value of the data item. Decimal alignment is controlled by the picture characters V and F.

Comma (,), point (.), or slash (/)
inserts a character into the associated position of the numeric character data when no zero suppression occurs. If zero suppression does occur, the character is inserted only under the following conditions:

- When an unsuppressed digit appears to the left of the character's position
- When a V appears immediately to the left of the character and the fractional part of the data item contains any significant digits
- When the character is at the start of the picture specification
- When the character is preceded only by characters that do not specify digit positions.

In all other cases where zero suppression occurs, a comma, point, or slash insertion character is treated as a zero suppression characters identical to the preceding character.

B specifies that a blank character be inserted into the associated position of the character value of the numeric character data.

Insertion and decimal point characters

The point, comma, or slash can be used in conjunction with the V to cause insertion of the point (or comma or slash) in the position that delimits the end of the integer portion in and the beginning of the fractional portion of a fixed-point (or floating-point) number, as might be desired in printing, since the V does not cause printing of a point. The point must immediately precede or immediately follow the V. If the point precedes the V, it is inserted only if an unsuppressed digit appears to the left of the V, even if all fractional digits are significant. If the point immediately follows the V, it is suppressed if all digits to the right of the V are suppressed, but it appears if there are any unsuppressed fractional digits (along with any intervening zeros).

The following example shows decimal conventions that are used in different countries.

```
declare A picture 'Z,ZZZ,ZZZV.99',
        B picture 'Z.ZZZ.ZZZV,99',
        C picture 'ZBZZZBZZZV,99';
A,B,C = 1234;
A,B,C = 1234.00;
```

A, B, and C represent nine-digit numbers with a decimal point or decimal comma assumed between the seventh and eighth digits. The actual point specified by the decimal point insertion character is not a part of the arithmetic value. It is, however, part of its character value. The two assignment statements assign the same character value to A, B, and C as follows:

```
1,234.00     /* value of A */
1.234,00     /* value of B */
1 234,00     /* value of C */
```

In the following example, decimal point alignment during assignment occurs on the character V. If Rate is printed, it appears as '762.00', but its arithmetic value is 7.6200.

```
declare Rate picture '9V99.99';
Rate = 7.62;
```

Figure 48 shows examples of insertion characters.

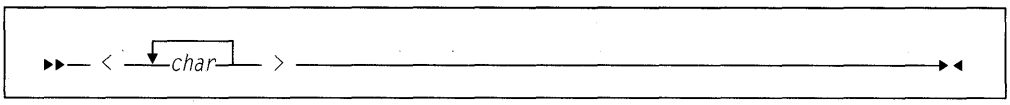
Figure 48. Examples of insertion characters

Source Attributes	Source Data (in constant form)	Picture Specification	Character Value
FIXED(4)	1234	9,999	1,234
FIXED(6,2)	1234.56	9,999V.99	1,234.56
FIXED(4,2)	12.34	ZZ.VZZ	12.34
FIXED(4,2)	00.03	ZZ.VZZ	␣␣␣03
FIXED(4,2)	00.03	ZZV.ZZ	␣␣.03
FIXED(4,2)	12.34	ZZV.ZZ	12.34
FIXED(4,2)	00.00	ZZV.ZZ	␣␣␣␣␣
FIXED(9,2)	12345667.89	9,999,999.V99	1,234,567.89
FIXED(7,2)	12345.67	**,999V.99	12,345.67
FIXED(7,2)	00123.45	**,999V.99	***123.45
FIXED(9,2)	1234567.89	9.999.999V,99	1.234.567,89
FIXED(6)	123456	99/99/99	12/34/56
FIXED(6)	101288	99-99-99	10-12-88
FIXED(6)	123456	99.9/99.9	12.3/45.6
FIXED(6)	001234	ZZ/ZZ/ZZ	␣␣␣12/34
FIXED(6)	000012	ZZ/ZZ/ZZ	␣␣␣␣␣12
FIXED(6)	000000	ZZ/ZZ/ZZ	␣␣␣␣␣␣␣
FIXED(6)	000000	**/**/**	********
FIXED(6)	000000	**B**B**	**␣**␣**
FIXED(6)	123456	99B99B99	12␣34␣56
FIXED(3)	123	9BB9BB9	1␣␣2␣␣3
FIXED(2)	12	9BB/9BB	1␣␣/2␣␣

Defining currency symbols

A currency symbol can be used as a picture character denoting a character value of numeric character data. This symbol can be the Dollar sign ($) or any symbol you choose. The symbol can be any sequence of characters enclosed in < and > characters.

The syntax is for generalized currency symbol is:



< indicates the start of the currency symbol It acts as an escape character. If you want to use the character <, you must specify <<.

char
is any character that will be part of your currency symbol(s).

> indicates the end of the currency symbol If you want to use the character >, you must specify <>.

More than one > indicates a drifting string (discussed on page 296).

Examples of general insertion strings include the following:

<DM> represents the Deutschemark

<Fr> represents the French Franc

<K$> represents the Khalistan Dollar

<Sur.f> represents the Surinam Guilder

<$> represents the Dollar sign

If the character < or > must be included in the sequence, it must be preceded by another <. Therefore, < acts as an escape character also.

The entire sequence enclosed in < > represents one "symbol" and therefore represents the character value for one numeric character. If the symbol needs to be represented as a drifting picture character, you specify > following the "< >" to represent each occurrence.

For example,

`Pic '<DM>>>.>>9,V99'`
represents a 10 character numeric picture, yielding 11 characters after assignment.

`Pic '<Sur.f>999,V99'`
represents a 7 character numeric picture, yielding 11 characters after assignment.

`Pic '<K$>>>,>>9.V99'`
represents a 10 character numeric picture, yielding 11 characters after assignment.

`Pic '<$>>>,>>9.V99'` represents a 10 character numeric picture, yielding 10 characters after assignment.

`Pic '$$$,$$9.V99'` has the same value as the previous picture specification.

More examples of currency symbol definition include the following:

```
dcl p pic'<DM>9.999,V99';
p = 1234.40;                          /* Yields 'DM1.234,40'   */

dcl p pic'<DM>9.999,V99';
p =   34.40;                          /* Yields 'DM   34,40'   */

dcl p pic'<DM>>.>>9,V99';
p = 1234.40;                          /* Yields 'DM1.234,40'   */

dcl p pic'<DM>>.>>9,V99';
p =   34.40;                          /* Yields '   DM34,40'   */

dcl p pic'9.999,V99<K&dollar>';
p = 1234.40;                          /* Yields '1.234,40K$'*/
```

In this chapter, the term *currency symbol* and the $ symbol refer to the dollar sign or any user-defined currency symbol.

Signs and currency symbols

The picture characters S, +, and – specify signs in numeric character data. The picture character $ (or the currency symbol) specifies a currency symbol in the character value of numeric character data. Only one type of sign character can appear in each field.

currency symbol
specifies the currency symbol.

In the following example:

```
dcl Price picture '$99V.99';
Price = 12.45;
```

The character value of `Price` is `'$12.45'`. Its arithmetic value is `12.45`.

For information on specifying a character as a currency symbol, refer to "Defining currency symbols" on page 294.

S specifies the plus sign character (+) if the data value is >=0; otherwise, it specifies the minus sign character (–). The rules are identical to those for the currency symbol.

In the following example:

```
dcl Root picture 'S999';
```

The value `50` is held as `'+050'`, the value `0` as `'+000'` and the value `-243` as `'-243'`.

+ specifies the plus sign character (+) if the data value is >=0; otherwise, it specifies a blank. The rules are identical to those for the currency symbol.

– specifies the minus sign character (–) if the data value is <0; otherwise, it specifies a blank. The rules are identical to those for the currency symbol.

Signs and currency symbols can be used in either a static or a drifting manner.

Static use: Static use specifies that a sign, a currency symbol, or a blank appears in the associated position. An S, +, or – used as a static character can appear to the right or left of all digits in the mantissa and exponent fields of a floating-point specification, and to the right or left of all digit positions of a fixed-point specification.

Drifting use: Drifting use specifies that leading zeros are to be suppressed. In this case, the rightmost suppressed position associated with the picture character will contain a sign, a blank, or a currency symbol (except that where all digit positions are occupied by drifting characters and the value of the data item is zero, the drifting character is not inserted).

A drifting character is specified by multiple use of that character in a picture field. The drifting character must be specified in each digit position through which it can drift. Drifting characters must appear in a sequence of the same drifting character, optionally containing a V and one of the insertion characters comma, point, slash, or B. Any of the insertion characters slash, comma, or point within or immediately following the string is part of the drifting string. The character B always causes insertion of a blank, wherever it appears. A V terminates the drifting string, except when the arithmetic value of the data item is zero; in that case, the V is ignored. A field of a picture specification can contain only one drifting string. A drifting string

cannot be preceded by a digit position nor can it occur in the same field as the picture characters * and Z.

The position in the data associated with the characters slash, comma, and point appearing in a string of drifting characters will contain one of the following:

- Slash, comma, or point if a significant digit appears to the left
- The drifting symbol, if the next position to the right contains the leftmost significant digit of the field
- Blank, if the leftmost significant digit of the field is more than one position to the right.

If a drifting string contains the drifting character *n* times, the string is associated with *n*-1 conditional digit positions. The position associated with the leftmost drifting character can contain only the drifting character or blank, never a digit. Two different picture characters cannot be used in a drifting manner in the same field.

If a drifting string contains a V within it, the V delimits the preceding portion as a subfield, and all digit positions of the subfield following the V must also be part of the drifting string that commences the second subfield.

In the case in which all digit positions after the V contain drifting characters, suppression in the subfield occurs only if all of the integer and fractional digits are zero. The resulting edited data item is then all blanks (except for any insertion characters at the start of the field). If there are any nonzero fractional digits, the entire fractional portion appears unsuppressed.

If, during or before assignment to a picture, the fractional digits of a decimal number are truncated so that the resulting value is zero, the sign inserted in the picture corresponds to the value of the decimal number prior to its truncation. Thus, the sign in the picture depends on how the decimal value was calculated.

Figure 49 on page 297 shows examples of signs and currency symbol characters.

Figure 49 (Page 1 of 2). Examples of signs and currency characters

Source Attributes	Source Data (in constant form)	Picture Specification	Character Value
FIXED(5,2)	123.45	$999V.99	$123.45
FIXED(5,2)	012.00	99$	12$
FIXED(5,2)	001.23	$ZZZV.99	$bb1.23
FIXED(5,2)	000.00	$ZZZV.ZZ	bbbbbbb
FIXED(1)	0	$$$.$$	bbbbbb
FIXED(5,2)	123.45	$$$9V.99	$123.45
FIXED(5,2)	001.23	$$$9V.99	bb$1.23
FIXED(2)	12	$$$,999	bbb$012
FIXED(4)	1234	$$$,999	b$1,234
FIXED(5,2)	2.45	SZZZV.99	+bb2.45
FIXED(5)	214	SS,SS9	bb+214
FIXED(5)	–4	SS,SS9	bbbb–4
FIXED(5,2)	–123.45	+999V.99	b123.45
FIXED(5,2)	–123.45	–999V.99	–123.45
FIXED(5,2)	123.45	999V.99S	123.45+

Figure 49 (Page 2 of 2). Examples of signs and currency characters

Source Attributes	Source Data (in constant form)	Picture Specification	Character Value
FIXED(5,2)	001.23	++B+9V.99	␣␣␣+1.23
FIXED(5,2)	001.23	---9V.99	␣␣␣1.23
FIXED(5,2)	-001.23	SSS9V.99	␣␣-1.23

Credit, debit, and zero replacement characters.

The picture characters CR, and DB cannot be used with any other sign characters in the same field.

The character pairs CR (credit) and DB (debit) specify the signs of real numeric character data items.

CR specifies that the associated positions will contain the letters CR if the value of the data is <0. Otherwise, the positions will contain two blanks. The characters CR can appear only to the right of all digit positions of a field.

DB specifies that the associated positions will contain the letters DB if the value of the data is <0. Otherwise, the positions will contain two blanks. The characters DB can appear only to the right of all digit positions of a field.

Y specifies that a zero in the specified digit position is replaced unconditionally by the blank character.

Figure 50 shows examples of credit, debit, overpunched, and zero replacement characters.

Figure 50. Examples of credit, debit, and zero replacement characters

Source Attributes	Source Data (in constant form)	Picture Specification	Character Value
FIXED(3)	-123	$Z.99CR	$1.23CR
FIXED(4,2)	12.34	$ZZV.99CR	$12.34␣␣
FIXED(4,2)	-12.34	$ZZV.99DB	$12.34DB
FIXED(4,2)	12.34	$ZZV.99DB	$12.34␣␣
FIXED(4)	1021	999I	102A
FIXED(4)	-1021	Z99R	102J
FIXED(4)	1021	99T9	10B1
FIXED(5)	00100	YYYYY	␣␣1␣␣
FIXED(5)	10203	9Y9Y9	1␣2␣3
FIXED(5,2)	000.04	YYYVY9	␣␣␣␣4

Exponent characters

The picture characters K and E delimit the exponent field of a numeric character specification that describes floating-point decimal numbers. The exponent field is the last field of a numeric character floating-point picture specification. The picture characters K and E cannot appear in the same specification.

K specifies that the exponent field appears to the right of the associated position. It does not specify a character in the numeric character data item.

E specifies that the associated position contains the letter E, which indicates the start of the exponent field.

The value of the exponent is adjusted in the character value so that the first significant digit of the first field (the mantissa) appears in the position associated with the first digit specifier of the specification (even if it is a zero suppression character).

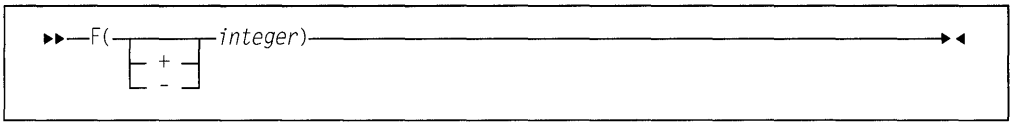
Figure 51 shows examples of exponent characters.

Figure 51. Examples of exponent characters

Source Attributes	Source Data (in constant form)	Picture Specification	Character Value
FLOAT(5)	.12345E06	V.99999E99	.12345E06
FLOAT(5)	.12345E–06	V.99999ES99	.12345E–06
FLOAT(5)	.12345E+06	V.99999KS99	.12345+06
FLOAT(5)	–123.45E+12	S999V.99ES99	–123.45E+12
FLOAT(5)	001.23E–01	SSS9.V99ESS9	+123.00E–3
FLOAT(5)	001.23E+04	ZZZV.99KS99	123.00+02
FLOAT(5)	001.23E+04	SZ99V.99ES99	+123.00E+02
FLOAT(5)	001.23E+04	SSSSV.99E–99	+123.00E02

Scaling factor

The picture character F specifies a picture scaling factor for fixed-point decimal numbers. It can appear only once at the right end of the picture specification. The syntax for F is:



F specifies the picture scaling factor. The picture scaling factor specifies that the decimal point in the arithmetic value of the variable is that number of places to the right (if the picture scaling factor is positive) or to the left (if negative) of its assumed position in the character value.

The number of digits following the V picture character minus the integer specified with F must be between -128 and 127.

Figure 52 shows examples of the picture scaling factor character.

Figure 52. Examples of scaling factor characters

Source Attributes	Source Data (in constant form)	Picture Specification	Character Value
FIXED(4,0)	1200	99F(2)	12
FIXED(7,0)	–1234500	S999V99F(4)	–12345
FIXED(5,5)	.00012	99F(–5)	12
FIXED(6,6)	.012345	999V99F(–4)	12345

Chapter 15. Condition handling

Chapter 15. Condition handling

While a PL/I program is running, a variety of events can occur that you can test for, respond to, and recover from. These events are called *conditions*, and are *raised* when detected. The conditions can be unexpected errors, such as overflow, an input/output transmission error, or the end of a page when output is sent to be printed but does not print. Conditions can also be expected (for example, the end of an input file). A condition is also raised when a SIGNAL statement for that condition is executed.

A condition is *enabled* when raising it will execute an on-unit or perform a system-defined action.

An action specified to be executed when an enabled condition is raised is *established*. You control condition handling by enabling conditions and specifying the required action for raised conditions.

The established action can be an ON-unit or the implicit action defined for the condition.

When an ON-unit is invoked, it is treated as a procedure without parameters. To assist you in making use of ON-units, built-in functions and pseudovariables are provided that you can use to inquire about the cause of a condition. They are listed in Chapter 17, "Built-in functions, pseudovariables, and subroutines" on page 371.

The implicit action for many conditions is to raise the ERROR condition. This provides a common condition that can be used to check for a number of different conditions, rather than checking each condition separately.

The condition handling built-in functions provide information such as the name of the entry point of the procedure in which the condition was raised, the character or character string (or graphic string) that raised a CONVERSION condition, the value of the key used in the last record transmitted, and so on. Some can be used as pseudovariables for error correction.

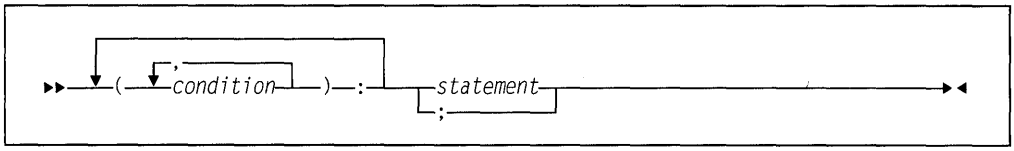
The ONCODE built-in function returns an integer indicating the cause of the last condition. ONCODE can be used to identify the specific circumstances that raise a particular condition (for example, the ERROR condition). The codes corresponding to the conditions and errors detected are listed in "Condition codes" on page 329.

Condition prefixes

You can specify whether or not some conditions are enabled or disabled. If a condition is enabled, the raising of the condition executes an action. If a condition is disabled, the raising of the condition does not execute an action.

Enabling and disabling can be specified for the eligible conditions by a condition prefix.

The syntax for the condition prefix is:



condition
Some conditions are always enabled, and cannot be disabled. Some are enabled unless you disable them, and some are disabled unless you enable them. The conditions are listed in Chapter 16, "Conditions" on page 312.

statement
Condition prefixes are not valid for DECLARE, DEFAULT, DO, SELECT, WHEN, OTHERWISE, END, IF, ELSE, and % statements. For information on the scope of condition prefixes, refer to "Scope of the condition prefix" on page 304.

In the following example `(size):` is the condition prefix. The conditional prefix indicates that the corresponding condition is enabled within the scope of the prefix.

```
(size):  L1:  X=(I**N) / (M+L);
```

Conditions can be enabled using the condition prefix specifying the condition name. They can be disabled using the condition prefix specifying the condition name preceded by NO without intervening blanks. Types and status of conditions are shown in Figure 53.

Figure 53 (Page 1 of 2). Classes and status of conditions

Class and conditions	Status
Computational (for data handling, expression evaluation, and computation)	
CONVERSION	Enabled by default
FIXEDOVERFLOW	Enabled by default
INVALIDOP	Enabled by default
OVERFLOW	Enabled by default
UNDERFLOW	Disabled by default
ZERODIVIDE	Enabled by default
Input/Output	
ENDFILE	Always enabled
ENDPAGE	Always enabled
KEY	Always enabled
NAME	Always enabled
RECORD	Always enabled
TRANSMIT	Always enabled
UNDEFINEDFILE	Always enabled

Figure 53 (Page 2 of 2). Classes and status of conditions

Class and conditions	Status
Program checkout (for debugging a program)	
SIZE	Disabled by default
STRINGRANGE	Disabled by default
STRINGSIZE	Disabled by default
SUBSCRIPTRANGE	Disabled by default
Miscellaneous	
AREA	Always enabled
ATTENTION	Always enabled
CONDITION	Always enabled
ERROR	Always enabled
FINISH	Always enabled
STORAGE	Always enabled

When an enabled condition is raised, it causes the established or implicit action to be performed.

When a condition is disabled, its raising causes no action; the program is unaware that the event has occurred.

For information about the performance effects of enabling and disabling conditions, refer to the *PL/I Package/2 Programming Guide*.

Scope of the condition prefix

The scope of a condition prefix (the part of the program to which it applies) is the statement or block to which the prefix is attached. The prefix does not necessarily apply to any procedures or ON-units that can be invoked in the execution of the statement.

A condition prefix attached to a PACKAGE, PROCEDURE, or BEGIN statement applies to all the statements up to and including the corresponding END statement. This includes other PROCEDURE or BEGIN statements nested within that block.

Condition status can be redefined within a block by attaching a prefix to statements within the block, including PROCEDURE and BEGIN statements (thus redefining the enabling or disabling of the condition within nested blocks). The redefinition applies only to the execution of the statement to which the prefix is attached. In the case of a nested PROCEDURE or BEGIN statement, it applies only to the block the statement defines, as well as any blocks contained within that block.

Raising conditions with OPTIMIZATION

When OPTIMIZATION is in effect, conditions for the same expression that appear multiple times may be raised only once. In the following example, SUBSCRIPTRANGE for IX may be raised only once:

```
Call P (55);
(subscriptrange): P: proc (IX);
  Dcl (Ar, Br, Cr) (10);
  Ar(IX) = Ar(IX) + Br(IX);
  t = Cr(IX);
End P;
```

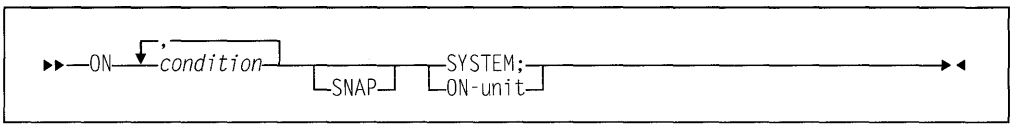
ON-units

An implicit action exists for every condition. When an enabled condition is raised, the implicit action is executed unless an ON-unit for the enabled condition is established.

ON statement

The ON statement establishes the action to be executed for any subsequent raising of an enabled condition in the scope of the established condition.

The syntax for the ON statement is:



condition
is any one of those described in Chapter 16, "Conditions" on page 312 or defined with the CONDITION attribute.

SNAP specifies that when the enabled condition is raised, diagnostic information relating to the condition is printed. The action of the SNAP option precedes the action of the ON-unit.

If SNAP and SYSTEM are specified, the implicit action is followed immediately by SNAP information.

SYSTEM
specifies that the implicit action is taken. The implicit action is not the same for every condition, although for most conditions a message is printed and the ERROR condition is raised. The implicit action for each condition is given in Chapter 16, "Conditions" on page 312.

ON-unit
specifies the action to be executed when the condition is raised and is enabled. The action is defined by the statement or statements in the ON-unit itself. The ON-unit is not executed at the time the ON statement is executed; it is executed only when the specified enabled condition is raised.

The ON-unit can be either a single unlabeled simple statement or an unlabeled begin block. If it is a simple statement, it can be any statement except BEGIN, DECLARE, DEFAULT, DO, END, FORMAT, ITERATE,

LEAVE, OTHERWISE, PROCEDURE, RETURN, SELECT, WHEN, or % statements. If the ON-unit is a begin block, a RETURN statement can appear only within a procedure nested within the begin block; a LEAVE statement can appear only within a do group nested within the begin block.

An ON-unit is treated as a procedure (without parameters) that is internal to the block in which it appears. Any names referenced in an ON-unit are those known in the environment in which the ON statement for that ON-unit was executed, rather than the environment in which the condition was raised.

When execution of the ON-unit is complete, control generally returns to the block from which the ON-unit was entered. Just as with a procedure, control can be transferred out of an ON-unit by a GO TO statement. In this case, control is transferred to the point specified in the GO TO, and a normal return does not occur.

The specific point to which control returns from an ON-unit varies for different conditions. Normal return for each condition is described in Chapter 16, "Conditions" on page 312.

Null ON-unit

The effect of a null statement ON-unit is to execute normal return from the condition.

Use of the null ON-unit is different from disabling a condition for two reasons:

- A null ON-unit can be specified for any condition, but not all conditions can be disabled.
- Disabling a condition, if possible, can save time by avoiding any checking for this condition. (If a null ON-unit is specified, the PL/I must still check for the raising of the condition.)

Scope of the ON-unit

The execution of an ON statement establishes an action specification for a condition. Once this action is established, it remains established throughout that block and throughout all dynamically descendent blocks until it is overridden by the execution of another ON statement or a REVERT statement or until termination of the block in which the ON statement is executed. (For information on dynamically descendent ON-units, refer to "Dynamically descendent ON-units" on page 307.)

When another ON statement specifies the same conditions:

- If a later ON statement specifies the same condition as a prior ON statement and this later ON statement is executed in a block which is a dynamic descendant of the block containing the prior ON statement, the action specification of the prior ON statement is temporarily suspended, or stacked. It can be restored either by the execution of a REVERT statement, or by the termination of the block containing the later ON statement.

  When control returns from a block, all established actions that existed at the time of its activation are reestablished. This makes it impossible for a subroutine to alter the action established for the block that invoked the subroutine.

- If the later ON statement and the prior ON statement are internal to the same invocation of the same block, the effect of the prior ON statement is logically nullified. No reestablishment is possible, except through execution of another ON statement (or re-execution of an overridden ON statement).

Dynamically descendent ON-units

It is possible to raise a condition during execution of an ON-unit that specifies another ON-unit. An ON-unit entered because a condition is either raised or signalled in another ON-unit is a dynamically descendent ON-unit. A normal return from a dynamically descendent ON-unit reestablishes the environment of the ON-unit in which the condition was raised.

A loop can occur if an ERROR condition raised in an ERROR ON-unit executes the same ERROR ON-unit, raising the ERROR condition again. To avoid a loop caused by this situation, use the following technique:

```
on error begin;
   on error system;
   ⋮
   end;
```

ON-units for file variables

An ON statement that specifies a file variable refers to the file constant that is the current value of the variable when the ON-unit is established.

Example 1

```
dcl  F file,
          G file variable;
          G = F;
L1:  on endfile(G);
L2:  on endfile(F);
```

The statements labeled L1 and L2 are equivalent.

Example 2

```
declare FV file variable,
        FC1 file,
        FC2 file;
FV = FC1;
on endfile(FV) go to Fin;
   ⋮
FV = FC2;
read file(FC1) into (X1);
read file(FV) into (X2);
```

An ENDFILE condition raised during the first READ statement causes the ON-unit to be entered, because the ON-unit refers to file FC1. If the condition is raised in the second READ statement, however, the ON-unit is not entered, because this READ refers to file FC2.

Example 3

```
E:  procedure;
  declare F1 file;
  on endfile (F1) goto L1;
  call E1 (F1);
  :
E1:  procedure (F2);
  declare F2 file;
  on endfile (F2) go to L2;
  read file (F1);
  read file (F2);
  end E1;
```

An end-of-file encountered for F1 in E1 causes the ON-unit for F2 in E1 to be entered. If the ON-unit in E1 was not specified, an ENDFILE condition encountered for either F1 or F2 would cause entry to the ON-unit for F1 in E.

Example 4

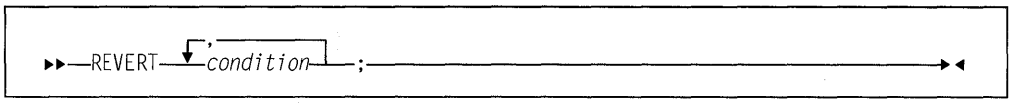
```
declare FV file variable,
        FC1 file,
        FC2 file;

do FV=FC1,FC2;
   on endfile(FV) go to Fin;
end;
```

If an ON statement specifying a file variable is executed more than once, and the variable has a different value each time, a different ON-unit is established at each execution.

REVERT statement

Execution of the REVERT statement in a given block cancels the ON-unit for the condition that executed in that block. The ON-unit that was established at the time the block was activated is then reestablished. REVERT affects only ON statements that are internal to the block in which the REVERT statement occurs and that have been executed in the same invocation of that block.

The syntax for the REVERT statement is:



condition
is any one of those described in Chapter 16, "Conditions" on page 312 or defined with the CONDITION attribute.

The REVERT statement cancels an ON-unit only if both of the following conditions are true:

1. An ON statement that is eligible for reversion, and that specifies a condition listed in the REVERT statement, was executed after the block was activated.

2. A REVERT statement with the specified condition was not previously executed in the same block.

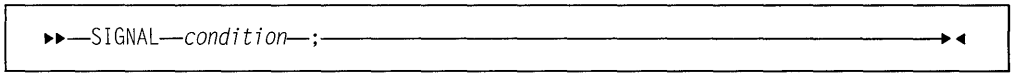
If either of these two conditions is not met, the REVERT statement is treated as a null statement.

SIGNAL statement

You can raise a condition by means of the SIGNAL statement. This statement can be used in program testing to verify the action of an ON-unit and to determine whether the correct action is associated with the condition. The established action is taken unless the condition is disabled.

If the specified condition is disabled, the SIGNAL statement becomes equivalent to a null statement.

The syntax for the SIGNAL statement is:

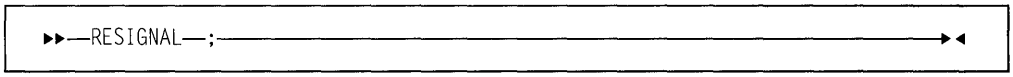


condition
is any condition described in Chapter 16, "Conditions" on page 312 or defined with the CONDITION attribute.

RESIGNAL statement

The RESIGNAL statement terminates the current ON-unit and allows another ON-unit to get control. The processing continues as if the ON-unit executing the RESIGNAL did not exist and was never given control. It allows multiple ON-units to get control for the same condition. It is valid only within an ON-unit or its dynamic descendants.

The syntax for RESIGNAL is:



Multiple conditions

A multiple condition is the simultaneous raising of two or more conditions.

The conditions for which a multiple condition can occur are:

RECORD, discussed on page 322
TRANSMIT, discussed on page 326.

The TRANSMIT condition is always processed first. The RECORD condition is ignored unless there is a normal return from the TRANSMIT ON-unit.

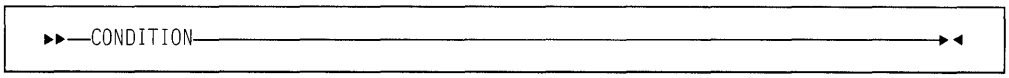
Multiple conditions are processed successively. When one of the following events occurs, no subsequent conditions are processed:

- Condition processing terminates the program, through implicit action for the condition, normal return from an ON-unit, or abnormal termination in the ON-unit.
- A GO TO statement transfers control from an ON-unit, so that a normal return is not allowed.

CONDITION attribute

The CONDITION attribute specifies that the declared name identifies a programmer-defined condition.

The syntax for the CONDITION attribute is:



A name that appears with the CONDITION condition in an ON, SIGNAL, or REVERT statement is contextually declared to be a condition name.

The default scope is EXTERNAL. An example of the CONDITION condition appears on page 314.

Chapter 16. Conditions

Chapter 16. Conditions

This chapter describes conditions in alphabetic order. In general, the following information is given for each condition:

- **Status**—an indication of the enabled/disabled status of the condition at the start of the program, and how the condition can be disabled (if possible) or enabled. Figure 53 on page 303 classifies the conditions into types, shows their status, and lists the conditions for disabling an enabled one.
- **Result**—the result of the operation that raised the condition. This applies when the condition is disabled as well as when it is enabled. In some cases, the result is undefined.
- **Cause and syntax**—A discussion of the condition, including the circumstances under which the condition can be raised. Raising conditions with the SIGNAL statement is discussed in "SIGNAL statement" on page 309.
- **Implicit action**—the action taken when an enabled condition is raised and no ON-unit is currently established for the condition.
- **Normal return**—the point to which control is returned as a result of the normal termination of the ON-unit. A GO TO statement that transfers control out of an ON-unit is an abnormal ON-unit termination. If a condition (except the ERROR condition) has been raised by the SIGNAL statement, the normal return is always to the statement immediately following SIGNAL.
- **Condition codes**—the codes corresponding to the conditions and errors for which the program is checked. An explanation for each code is given under "Condition codes" on page 329.

AREA condition

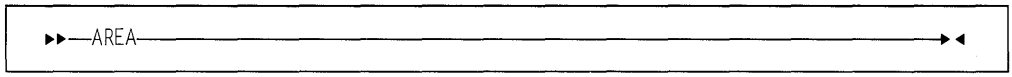
Status: AREA is always enabled.

Result: An attempted allocation or assignment that raises the AREA condition has no effect.

Cause and syntax: The AREA condition is raised in either of the following circumstances:

- When an attempt is made to allocate a based variable within an area that contains insufficient free storage for the allocation to be made.
- When an attempt is made to perform an area assignment, and the target area contains insufficient storage to accommodate the allocations in the source area.

The syntax for AREA is:



Implicit action: A message is printed and the ERROR condition is raised.

Normal return: On normal return from the ON-unit, the action is as follows:

- If the condition was raised by an allocation and the ON-unit is a null ON-unit, the allocation is not attempted again.
- If the condition was raised by an allocation, the allocation is attempted again. Before the attempt is made, the area reference is reevaluated. Thus, if the ON-unit has changed the value of a pointer qualifying the reference to the inadequate area so that it points to another area, the allocation is attempted again within the new area.
- If the condition was raised by an area assignment, or by a SIGNAL statement, execution continues from the point at which the condition was raised.

Condition codes: 360, 361, 362

ATTENTION condition

Status: ATTENTION is always enabled.

Result: Raising the condition causes an ATTENTION ON-unit to be entered. If there is no ATTENTION ON-unit, the condition is ignored, and there is no change in the flow of control.

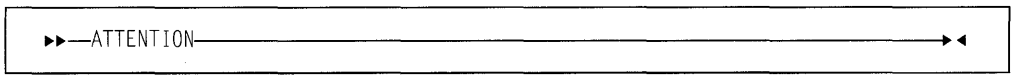
Cause and syntax: The ATTENTION condition is raised when the user hits CTRL-BRK or CTRL-C to interrupt a non-Presentation Manager application. (The Presentation Manager does not recognize CTRL-BRK and CTRL-C.) The condition can also be raised by a SIGNAL ATTENTION statement.

The MAIN procedure must be a PL/I procedure compiled with the INTERRUPT option. The runtime option INTERRUPT must be also be set to INTERRUPT(ON) to enable the ATTENTION interrupt capability. If the runtime option INTERRUPT(ON) is not set, use of the INTERRUPT option to enable the ATTENTION interrupt capability is invalid.

An ATTENTION ON-unit is entered when:

- The environment passes an interrupt request to the application program, and the program was compiled using the INTERRUPT option.
- A SIGNAL ATTENTION statement is executed. In this case the compiler INTERRUPT option is not required.

The syntax for ATTENTION is:



Abbreviation: ATTN

Implicit action: The attention is ignored.

Normal return: On return from an ATTENTION ON-unit, processing is resumed at a point in the program immediately following the point at which the condition was raised.

Condition code: 400

CONDITION condition

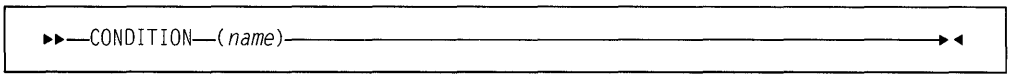
Status: CONDITION is always enabled.

Result: The CONDITION condition allows you to establish an ON-unit that will be executed whenever a SIGNAL statement for the appropriate CONDITION condition is executed.

As a debugging aid, the CONDITION condition can be used to establish an ON-unit that prints information about the current status of the program.

Cause and syntax: The CONDITION condition is raised by a SIGNAL statement. The name specified in the SIGNAL statement determines which CONDITION condition is raised. The ON-unit can be executed from any point in the program through placement of a SIGNAL statement. Normal rules of name scope apply. A condition name is external by default, but can be declared INTERNAL.

The syntax for CONDITION is:



Abbreviation: COND

The following example shows the use of the CONDITION condition.

```
dcl Test condition;

on condition (Test)
  begin;
  :
  end;
```

The begin block is executed whenever the following statement is executed:

```
signal condition (Test);
```

Implicit action: A message is printed and execution continues with the statement following SIGNAL.

Normal return: Execution continues with the statement following the SIGNAL statement.

Condition code: 500

CONVERSION condition

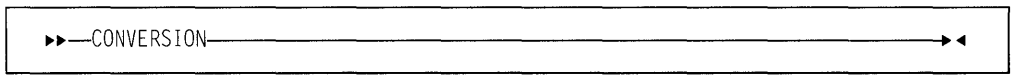
Status: CONVERSION is enabled throughout the program, except within the scope of the NOCONVERSION condition prefix. You can use the ONSOURCE, ONCHAR, and ONGSOURCE pseudovariables in CONVERSION ON-units to correct conversion errors.

Result: When CONVERSION is raised, the contents of the entire result field are undefined.

Cause and syntax: The CONVERSION computational condition is raised whenever an invalid conversion is attempted on character data. This attempt can be made internally or during an input/output operation. For example, the condition is raised when:

- A character other than 0 or 1 exists in character data being converted to bit data.
- A character value being converted to a numeric character field, or to a coded arithmetic value, contains characters which are not the representation of an optionally signed arithmetic constant, or an expression to represent a complex constant.
- A value being converted to a character pictured item contains characters not allowed by the picture specification.

The syntax for CONVERSION is:



Abbreviation: CONV

All conversions of character data are carried out character-by-character in a left-to-right sequence. The condition is raised for each invalid character. The condition is also raised if all the characters are blank, with the following exceptions:

- For input with the F-format item, a value of zero is assumed
- For input with the E-format item, be aware that sometimes the ON-unit will be repeatedly entered.

When an invalid character is encountered, the current action specification for the condition is executed (provided that CONVERSION is not disabled). If the action specification is an ON-unit, the invalid character can be replaced within the unit. For nongraphic source data, use the ONSOURCE or ONCHAR pseudovariables. For graphic source data, use the ONGSOURCE pseudovariable.

If the CONVERSION condition is raised and it is disabled, the program is in error.

If the CONVERSION condition is raised during conversion from graphic data to nongraphic data, the ONCHAR and ONSOURCE built-in functions do not contain valid source data. The ONGSOURCE built-in function contains the original graphic source data. The graphic conversion will be retried if the ONGSOURCE pseudovariable is used in the CONVERSION ON-unit to attempt to fix the graphic data that raised the CONVERSION condition. If the ONGSOURCE pseudovariable is not used in the CONVERSION ON-unit, the ERROR condition will be raised.

Implicit action: A message is printed and the ERROR condition is raised.

Normal return: If CONVERSION was raised on a character string source (not graphic source) and either ONSOURCE or ONCHAR pseudovariables are used in the ON-unit, the program retries the conversion on return from the ON-unit.

If CONVERSION was raised on a graphic source and the ONGSOURCE pseudovariable is used in the ON-unit, the program retries the conversion on return from the ON-unit.

If the conversion error is not corrected using these pseudovariables, the program will loop.

Condition codes: 600-666

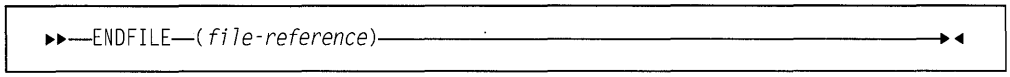
ENDFILE condition

Status: The ENDFILE condition is always enabled.

Result: If the specified file is not closed after the condition is raised, subsequent GET or READ statements to the file are unsuccessful and cause additional ENDFILE conditions to be raised.

Cause and syntax: The ENDFILE input/output condition can be raised during a operation by an attempt to read past the end of the file specified in the GET or READ statement. It applies only to SEQUENTIAL INPUT, SEQUENTIAL UPDATE, and STREAM INPUT files.

The syntax for ENDFILE is:



In record-oriented data transmission, ENDFILE is raised whenever an end of file is encountered during the execution of a READ statement.

In stream-oriented data transmission, ENDFILE is raised during the execution of a GET statement if an end of file is encountered either before any items in the GET statement data list have been transmitted or between transmission of two of the data items. If an end of file is encountered while a data item is being processed, or if it is encountered while an X-format item is being processed, the ERROR condition is raised.

Implicit action: A message is printed and the ERROR condition is raised.

Normal return: Execution continues with the statement immediately following the GET or READ statement that raised the ENDFILE.

If a file is closed in an ON-unit for this condition, the results of normal return are undefined. Exit from the ON-unit with the closed file must be achieved with a GO TO statement.

Condition code: 70

ENDPAGE condition

Status: ENDPAGE is always enabled.

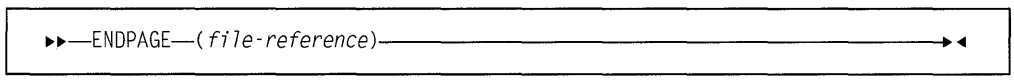
Result: When ENDPAGE is raised, the current line number is one greater than that specified by the PAGESIZE option (default is 61) so that it is possible to continue writing on the same page. The ON-unit can start a new page by execution of a PAGE option or a PAGE format item, which sets the current line to one.

If the ON-unit does not start a new page, the current line number can increase indefinitely. If a subsequent LINE option or LINE format item specifies a line number that is less than or equal to the current line number, ENDPAGE is not raised, but a new page is started with the current line set to one. An exception is that if the current line number is equal to the specified line number, and the file is positioned on column one of the line, ENDPAGE is not raised.

If ENDPAGE is raised during data transmission, on return from the ON-unit, the data is written on the current line, which might have been changed by the ON-unit. If ENDPAGE results from a LINE or SKIP option, on return from the ON-unit, the action specified by LINE or SKIP is ignored.

Cause and syntax: The ENDPAGE input/output condition is raised when a PUT statement results in an attempt to start a new line beyond the limit specified for the current page. This limit can be specified by the PAGESIZE option in an OPEN statement; if PAGESIZE has not been specified, a default limit of 60 is applied. The attempt to exceed the limit can be made during data transmission (including associated format items, if the PUT statement is edit-directed), by the LINE option, or by the SKIP option. ENDPAGE can also be raised by a LINE option or LINE format item that specified a line number less than the current line number. ENDPAGE is raised only once per page, except when it is raised by the SIGNAL statement.

The syntax for ENDPAGE is:



Implicit action: A new page is started. If the condition is signaled, execution is unaffected and continues with the statement following the SIGNAL statement.

Normal return: Execution of the PUT statement continues in the manner described above.

Condition code: 90

ERROR condition

Status: ERROR is always enabled.

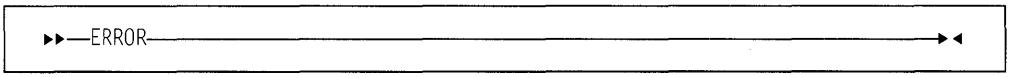
Result An error message is issued if no ON-unit is active when the ERROR condition arises or if the ON-unit does not use a GOTO (to exit the block) to recover from the condition.

Cause and syntax: The ERROR condition is the implicit action for many conditions. This provides a common condition that can be used to check for a number of different conditions, rather than checking each condition separately.

The ERROR condition is raised under the following circumstances:

- As a result of the implicit action for a condition, which is to raise the ERROR condition.
- As a result of the normal return action for some conditions, such as SUBSCRIPTRANGE CONVERSION or when no retry is attempted.
- As a result of an error (for which there is no other PL/I-defined condition) during program execution.

The syntax for ERROR is:



Implicit action

The FINISH condition is raised.

Normal return: The implicit action is taken.

Condition codes: All codes 1000 and above are ERROR conditions.

FINISH condition

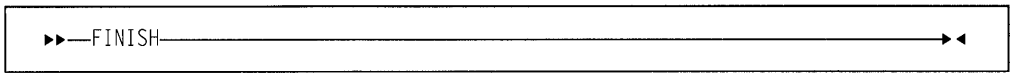
Status: FINISH is always enabled.

Result: Control passes to the FINISH ON-unit and processing continues.

Cause and syntax: The FINISH condition is raised during execution of a statement that would terminate the procedures. The following actions take place:

- If the termination is normal:
  - The FINISH On-Unit, if established, is given control only if the main procedure is PL/I.
- If the termination is abnormal:
  - The FINISH On-Unit, if established in an active block, is given control.

The syntax for FINISH is:



Implicit action: No action is taken and processing continues from the point where the condition was raised.

Normal return: Processing resumes at the point where the condition was raised. This point is the statement following the SIGNAL statement if the conditions was signalled.

Condition code: 4

FIXEDOVERFLOW condition

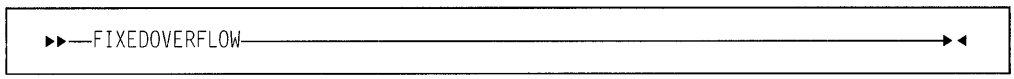
Status: FIXEDOVERFLOW is enabled throughout the program, except within the scope of the NOFIXEDOVERFLOW condition prefix.

Result: The result of the invalid fixed-point operation is undefined.

Cause and syntax: The FIXEDOVERFLOW computational condition is raised when the length of the result of a fixed-point arithmetic operation exceeds the maximum length allowed by the implementation.

The FIXEDOVERFLOW condition differs from the SIZE condition in that SIZE is raised when a result exceeds the declared size of a variable, while FIXEDOVERFLOW is raised when a result exceeds the maximum allowed by the computer.

The syntax for FIXEDOVERFLOW is:



Abbreviation: FOFL

If the FIXEDOVERFLOW condition is raised and it is disabled, the program is in error.

Implicit action: A message is printed and the ERROR condition is raised.

Normal return: Control returns to the point immediately following the point at which the condition was raised.

Condition code: 310

Note: If the SIZE condition is disabled, an attempt to assign an oversize number to a fixed decimal variable can raise the FIXEDOVERFLOW condition.

Because checking the result lengths requires a substantial overhead in both storage space and run time, the FIXEDOVERFLOW condition is primarily used for program testing. It can be removed for production programs. For more information

on testing and production application programs, refer to the *PL/I Package/2 Programming Guide*.

INVALIDOP condition

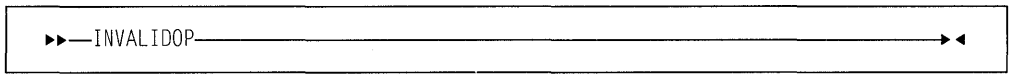
Status: INVALIDOP is enabled throughout the program, except within the scope of the NOINVALIDOP condition prefix.

Result: The result of the invalid operation is undefined.

Cause and syntax: The INVALIDOP computational condition is raised when any of the following are detected during the evaluation of floating-point expressions.

- Subtraction of two infinities
- Multiplication of infinity by 0
- Division of two infinities
- Division of zero by zero
- 387 coprocessor invalid operations
- 387 coprocessor stack faults
- Other 387 coprocessor exceptions.

The syntax for INVALIDOP is:



Implicit action: The ERROR condition is raised.

Normal return: Control returns to the point immediately following the point at which the condition was raised. If INVALIDOP was not signalled, continued use of the data related to the invalid operation can raise additional conditions. If a 387 hardware exception occurs, continued program execution can raise additional conditions.

Condition code: 290

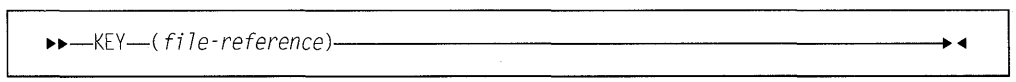
KEY condition

Status: KEY is always enabled.

Result: The keyed record is undefined, and the statement in which it appears is ignored.

Cause and syntax: The KEY input/output condition is raised when a record with a specified key cannot be found. The condition can be raised only during operations on keyed records. It is raised for the condition codes listed below.

The syntax for KEY is:



When a LOCATE statement is used for data set, the KEY condition for this LOCATE statement is not raised until the next WRITE or LOCATE statement for the file, or when the file is closed.

Implicit action: A message is printed and the ERROR condition is raised.

Normal return: Control passes to the statement immediately following the statement that raised KEY.

If a file is closed in an ON-unit for this condition, the results of normal return are undefined. Exit from the ON-unit with the closed file must be achieved with a GO TO statement.

Condition codes: 50-58

NAME condition

Status: NAME is always enabled.

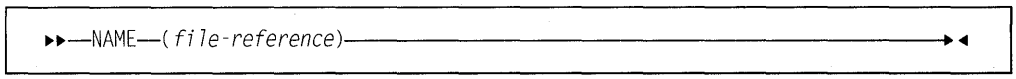
Result: The named data is undefined.

Cause and syntax: The NAME input/output condition can be raised only during execution of a data-directed GET statement with the FILE option. It is raised in any of the following situations:

- The syntax is not correct, as described under "Syntax of data-directed data" on page 260.
- The name is missing or invalid:
  - No counterpart is found in the data list.
  - If there is no data list, the name is not known in the block.
  - A qualified name is not fully qualified.
  - More than 256 characters have been specified for a fully qualified name.
  - DBCS contains a byte outside the valid range X'41' to X'FE'.
- A subscript list is missing or invalid:
  - A subscript is missing.
  - The number of subscripts is incorrect.
  - More than 10 digits are in a subscript (leading zeros ignored).
  - A subscript is outside the allowed range of the current allocation of the variable.

You can retrieve the incorrect data field by using the built-in function DATAFIELD in the ON-unit.

The syntax for NAME is:



Implicit action: The incorrect data field is ignored, a message is printed, and execution of the GET statement continues.

Normal return: The execution of the GET statement continues with the next name in the stream.

Condition code: 10

OVERFLOW condition

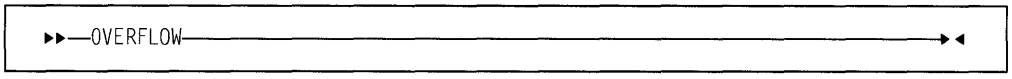
Status: OVERFLOW is enabled throughout the program, except within the scope of the NOOVERFLOW condition prefix.

Result: The value of such an invalid floating-point number is undefined.

Cause and syntax: The OVERFLOW computational condition is raised when the magnitude of a floating-point number exceeds the maximum allowed.

The OVERFLOW condition differs from the SIZE condition in that SIZE is raised when a result exceeds the declared size of a variable, while OVERFLOW is raised when a result exceeds the maximum allowed by the computer.

The syntax for OVERFLOW is:



Abbreviation: OFL

If the OVERFLOW condition is raised and it is disabled, the program is in error.

Implicit action: A message is printed and the ERROR condition is raised.

Normal return: Control returns to the point immediately following the point at which the condition was raised.

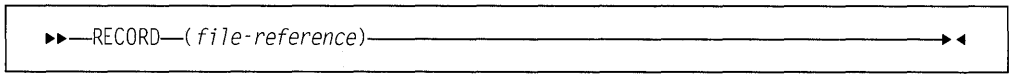
Condition code: 300

RECORD condition

Status: RECORD is always enabled.

Result: The length prefix for the specified file can be inaccurately transmitted.

Cause and syntax: The RECORD input/output condition is raised if the specified record is truncated. The condition can be raised only during a READ, WRITE, LOCATE, or REWRITE operation.



If the SCALARVARYING option is applied to the file (It must be applied to a file using locate mode to transmit varying-length strings.), a 2-byte length prefix is transmitted with an element varying-length string. The length prefix is not reset if the RECORD condition is raised. If the SCALARVARYING option is not applied to the file, the length prefix is not transmitted. On input, the current length of a

varying-length string is set to the shorter of the record length and the maximum length of the string.

Implicit action: A message is printed and the ERROR condition is raised.

Normal return: Execution continues with the statement immediately following the one for which RECORD was raised.

If a file is closed in an ON-unit for this condition, the results of normal return are undefined. Exit from the ON-unit with the closed file must be achieved with a GO TO statement.

Condition codes: 20-24

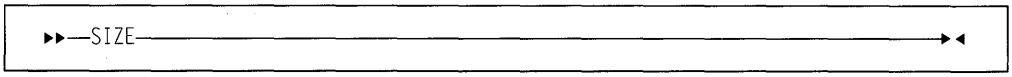
SIZE condition

Status: SIZE is disabled throughout the program, except within the scope of the SIZE condition prefix.

Result: The result of the assignment is undefined.

Cause and syntax: The SIZE computational condition is raised only when high-order (that is, leftmost) significant binary or decimal digits are lost in an attempted assignment to a variable or an intermediate result or in an input/output operation. This loss can result from a conversion involving different data types, different bases, different scales, or different precisions. The size condition is not enabled unless it appears in a condition prefix.

The syntax for SIZE is:



SIZE is raised when the size of the value being assigned to a data item exceeds the declared (or default) size of the data item, even if this is not the actual size of the storage that the item occupies. For example, a fixed binary item of precision (20) will occupy a fullword in storage, but SIZE is raised if a value whose size exceeds FIXED BINARY(20) is assigned to it.

Because checking sizes requires substantial overhead in both storage space and run time, the SIZE condition is primarily used for program testing. It can be removed from production programs. For more information on test and production application programs, refer to the *PL/I Package/2 Programming Guide*.

The SIZE condition differs from the FIXEDOVERFLOW condition in that FIXEDOVERFLOW is raised when the size of a calculated fixed-point value exceeds the maximum allowed by the implementation. SIZE can be raised on assignment of a value regardless of whether or not FIXEDOVERFLOW was raised in the calculation of that value.

If the SIZE condition is raised and it is disabled, the program is in error.

Implicit action: A message is printed and the ERROR condition is raised.

Normal return: Control returns to the point immediately following the point at which the condition was raised.

Condition codes: 340, 341

STORAGE condition

Status: STORAGE is always enabled.

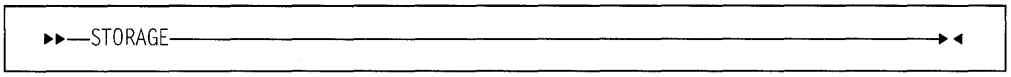
Result: The result depends on the type of variable for which attempted storage allocation raised the condition.

- After an ALLOCATE statement for a controlled variable, that variable's generation is not allocated. A reference to that controlled variable results in accessing the generation (if any) before the ALLOCATE statement.
- After an ALLOCATE statement for a based variable, the variable is not allocated and its associated pointer is undefined.

Cause and syntax: The STORAGE condition allows the program to gain control for the failure of an ALLOCATE statement that attempted to allocate BASED or CONTROLLED storage outside of an AREA. Failure of an ALLOCATE statement in an AREA raises the AREA condition.

The ON-unit for the STORAGE condition can attempt to free allocated storage. If the ON-unit is unable to free sufficient storage, it can provide the necessary steps to terminate the program without losing diagnostic information.

The syntax for STORAGE is:



Implicit action: The ERROR condition is raised.

Normal return Control returns to the point immediately following the point at which the condition was raised. If the STORAGE condition was not signalled, but was caused by an ALLOCATE statement, the ALLOCATE statement that failed is retried.

Condition codes: 450, 451

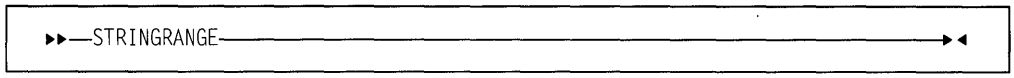
STRINGRANGE condition

Status: STRINGRANGE is disabled throughout the program, except within the scope of the STRINGRANGE condition prefix.

Result: The value of the specified SUBSTR is altered.

Cause and syntax: The STRINGRANGE program-checkout condition is raised whenever the values of the arguments to a SUBSTR reference fail to comply with the rules described for the SUBSTR built-in function. It is raised for each reference to an invalid argument.

The syntax for STRINGRANGE is:



Abbreviation: STRG

Implicit action: A message is printed and processing continues as described for normal return.

Normal return: Execution continues with a revised SUBSTR reference. for which the value is defined as follows:

Assuming that the length of the source string (after execution of the ON-unit, if specified) is *k*, the starting point is *i*, and the length of the substring is *j*;

- If *i* is greater than *k*, the value is the null string.
- If *i* is less than or equal to *k*, the value is that substring beginning at the *m*th character, bit, or graphic of the source string and extending *n* characters, bits, or graphics, where *m* and *n* are defined by:

  ```
  M = max( I,1 )
  N = max( 0,min( J + min(I,1) - 1,K - M + 1 ))
  ```

  if J is specified.

  ```
  N = K - M + 1
  ```

  if J is not specified.

This means that the new arguments are forced within the limits.

The values of *i* and *j* are established before entry to the ON-unit. They are not reevaluated on return from the ON-unit.

The value of *k* might change in the ON-unit if the first argument of SUBSTR is a varying-length string. The value *n* is computed on return from the ON-unit using any new value of *k*.

Condition code: 350

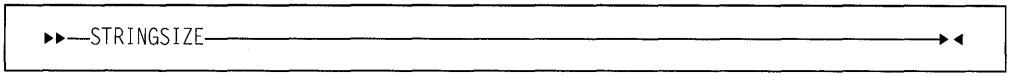
STRINGSIZE condition

Status: STRINGSIZE is disabled throughout the program, except within the scope of the STRINGSIZE condition prefix.

Result: After the condition action, the truncated string is assigned to its target string. The right-hand characters, bits, or graphics of the source string are truncated so that the target string can accommodate the source string.

Cause and syntax: The STRINGSIZE program-checkout condition is raised when you attempt to assign a string to a target with a shorter maximum length.

The syntax for STRINGSIZE is:



Abbreviation: STRZ

Implicit action: A message is printed and processing continues.

Normal return: Execution continues from the point at which the condition was raised.

Condition codes: 150, 151

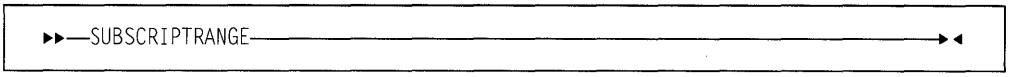
SUBSCRIPTRANGE condition

Status: SUBSCRIPTRANGE is disabled throughout the program, except within the scope of the SUBSCRIPTRANGE condition prefix.

Result: When SUBSCRIPTRANGE has been raised, the value of the invalid subscript is undefined, and, hence, the reference is also undefined.

Cause and syntax: The SUBSCRIPTRANGE program-checkout condition is raised whenever a subscript is evaluated and found to lie outside its specified bounds. The order of raising SUBSCRIPTRANGE relative to evaluation of other subscripts is undefined.

The syntax for SUBSCRIPTRANGE is:



Abbreviation: SUBRG

Implicit action: A message is printed and the ERROR condition is raised.

Normal return: Normal return from a SUBSCRIPTRANGE ON-unit raises the ERROR condition.

Condition codes: 520

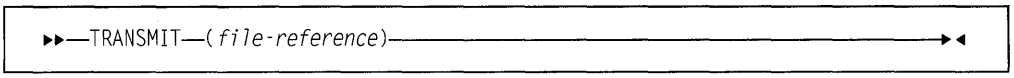
TRANSMIT condition

Status: TRANSMIT is always enabled.

Result Raising the TRANSMIT condition indicates that any data transmitted is potentially incorrect.

Cause and syntax: The TRANSMIT input/output condition can be raised during any input/output operation. It is raised by an uncorrectable transmission error of a record (or of a block, if records are blocked), which is an input/output error that could not be corrected during execution. It can be caused by a damaged recording medium, or by incorrect specification or setup.

The syntax for TRANSMIT is:



During input, TRANSMIT is raised after transmission of the potentially incorrect record. If records are blocked, TRANSMIT is raised for each subsequent record in the block.

During output, TRANSMIT is raised after transmission. If records are blocked, transmission will occur when the block is complete rather than after each output statement.

When a spanned record is being updated, the TRANSMIT condition is raised on the last segment of a record only. It is not raised for any subsequent records in the same block, although the integrity of these records cannot be assumed.

Implicit action: A message is printed and the ERROR condition is raised.

Normal return: Processing continues as though no error had occurred, allowing another condition (for example, RECORD) to be raised by the statement or data item that raised the TRANSMIT condition.

If a file is closed in an ON-unit for this condition, the results of normal return are undefined. Exit from the ON-unit with the closed file must be achieved with a GO TO statement.

Condition codes: 40-46

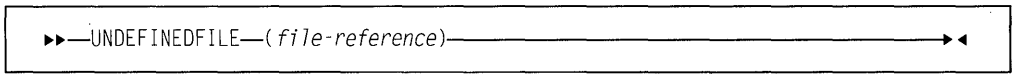
UNDEFINEDFILE condition

Status: UNDEFINEDFILE is always enabled.

Result: Specified files are undefined to the application program.

Cause and syntax: The UNDEFINEDFILE input/output condition is raised whenever an unsuccessful attempt to open a file is made. If the attempt is made by means of an OPEN statement that specifies more than one file, the condition is raised after attempts to open all specified files.

The syntax for UNDEFINEDFILE is:



Abbreviation: UNDF

If UNDEFINEDFILE is raised for more than one file in the same OPEN statement, ON-units are executed according to the order of appearance (taken from left to right) of the file names in that OPEN statement.

If UNDEFINEDFILE is raised by an implicit file opening in a data transmission statement, upon normal return from the ON-unit, processing continues with the

remainder of the data transmission statement. If the file was not opened in the ON-unit, the statement cannot continue and the ERROR condition is raised.

The UNDEFINEDFILE condition is raised not only by conflicting attributes (such as DIRECT with PRINT), but also by:

- Block size smaller than record size (except when records are spanned)
- LINESIZE exceeding the maximum allowed
- KEYLENGTH zero or not specified for creation of INDEXED data sets
- Specifying a KEYLOC option, for an INDEXED data set, with a value resulting in KEYLENGTH + KEYLOC exceeding the record length
- Specifying a V-format logical record length of less than 18 bytes for STREAM data sets
- Specifying a block size that is not an integral multiple of the record size for FB-format records
- Specifying a logical record length that is not at least 4 bytes smaller than the specified block size for VB-format records.

Implicit action: A message is printed and the ERROR condition is raised.

Normal return: Upon the normal completion of the final ON-unit, control is given to the statement immediately following the statement that raised the condition.

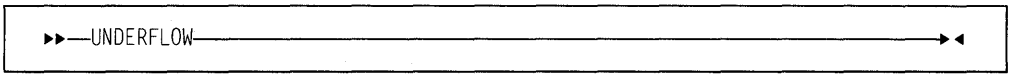
Condition codes: 80-89, 91-95

UNDERFLOW condition

Status: UNDERFLOW is enabled throughout the program, except within the scope of the NOUNDERFLOW condition prefix.

Result: The invalid floating-point value is set to 0.

Cause and syntax: The UNDERFLOW computational condition is raised when the magnitude of a floating-point number is smaller than the minimum allowed.



Abbreviation: UFL

UNDERFLOW is not raised when equal numbers are subtracted (often called significance error).

The expression $X^{(-Y)}$ (where Y>0) can be evaluated by taking the reciprocal of X^{Y}; hence, the OVERFLOW condition might be raised instead of the UNDERFLOW condition.

Implicit action: A message is printed, and execution continues from the point at which the condition was raised.

Normal return: Control returns to the point immediately following the point at which the condition was raised.

Condition code: 330

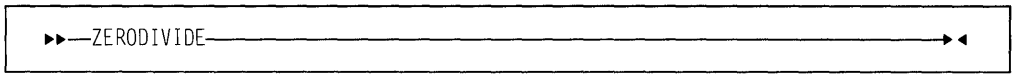
ZERODIVIDE condition

Status: ZERODIVIDE is enabled throughout the program, except within the scope of the NOZERODIVIDE condition prefix.

Result: The result of a division by zero is undefined.

Cause and syntax: The ZERODIVIDE computational condition is raised when an attempt is made to divide by zero. This condition is raised for fixed-point and floating-point division. If the numerator of a floating-point divide is also zero, INVALIDOP is raised.

The syntax for ZERODIVIDE is:



Abbreviation: ZDIV

If the ZERODIVIDE condition is raised and it is disabled, the program is in error.

Implicit action: A message is printed and the ERROR condition is raised.

Normal return: Control returns to the point immediately following the point at which the condition was raised.

Condition code: 320

Condition codes

Condition codes listed in this section reflect an aggregate of condition codes generated by all implementations. Some might not be generated for a particular platform.

The following is a summary of all condition codes in numerical sequence.

3 This condition is raised if, in a SELECT group, no *WHEN* clause is selected and no *OTHERWISE* clause is present.

4 SIGNAL FINISH, or STOP statement executed.

9 SIGNAL ERROR statement executed.

10 SIGNAL NAME statement executed.

20 SIGNAL RECORD statement executed.

21 Record variable smaller than record size. Either:

- The record is larger than the variable in a READ INTO statement; the remainder of the record is lost.
- The record length specified for a file with fixed-length records is larger than the variable in a WRITE, REWRITE, or LOCATE statement; the

remainder of the record is undefined. If the variable is a varying-length string, RECORD is not raised if the SCALARVARYING option is applied to the file.

22 Record variable larger than record size. Either:

- The record length specified for a file with fixed-length records is smaller than the variable in a READ INTO statement; the remainder of the variable is undefined. If the variable is a varying-length string, RECORD is not raised if the SCALARVARYING option is applied to the file.
- The maximum record length is smaller than the variable in a WRITE, REWRITE, or LOCATE statement. For WRITE or REWRITE, the remainder of the variable is lost; for LOCATE, the variable is not transmitted.
- The variable in a WRITE or REWRITE statement indicates a zero length; no transmission occurs. If the variable is a varying-length string, RECORD is not raised if the SCALARVARYING option is applied to the file.

23 Record variable length is either zero or too short to contain the embedded key.

The variable in a WRITE or REWRITE statement is too short to contain the data set embedded key; no transmission occurs. (This case currently applies only to indexed key-sequenced data sets.)

24 Zero length record was read from a REGIONAL data set.

40 SIGNAL TRANSMIT statement executed.

41 Uncorrectable transmission error in output data set.

42 Uncorrectable transmission error in input data set.

43 Uncorrectable transmission error on output to index set.

44 Uncorrectable transmission error on input from index set.

45 Uncorrectable transmission error on output to indexed consecutive data set.

46 Uncorrectable transmission error on input from consecutive data set.

50 SIGNAL KEY statement executed.

51 Key specified cannot be found.

52 Attempt to add keyed record that has same key as a record already present in data set; or, in a REGIONAL(1) data set, attempt to write into a region already containing a record.

53 Value of expression specified in KEYFROM option during sequential creation of INDEXED or REGIONAL data set is less than value of previously specified key or region number.

54 Key conversion error, possibly due to region number not being numeric character.

55 Key specification is null string or begins (8)'1'B or a change of embedded key has occurred on a sequential REWRITE[FROM] for an INDEXED or key-sequenced data set.

56 Attempt to access a record using a key that is outside the data set limits.

57 No space available to add a keyed record on INDEXED insert.

58 Key of record to be added lies outside the range(s) specified for the data set.

70 SIGNAL ENDFILE statement executed.

80 SIGNAL UNDEFINEDFILE statement executed.

81 Conflict in file attributes exists at open time between attributes in DECLARE statement and those in explicit or implicit OPEN statement.

82 Conflict between file attributes and physical organization of data set (for example, between file organization and device type), or indexed data set has not been loaded.

83 After merging ENVIRONMENT options with DD statement and data set label, data set specification is incomplete; for example, block size or record format has not been specified.

84 No DD statement associating file with a data set.

85 During initialization of a DIRECT OUTPUT file associated with a REGIONAL data set, an input/output error occurred.

86 LINESIZE greater than implementation-defined maximum, or invalid value in an ENVIRONMENT option.

87 After merging ENVIRONMENT options with DD statement and data set label, conflicts exist in data set specification; the value of LRECL, BLKSIZE or RECSIZE are incompatible with one another or the DCB FUNCTION specified.

88 After merging ENVIRONMENT options with DD statement and data set label, conflicts exist in data set specification; the resulting combination of MODE/FUNCTION and record format are invalid.

89 Password invalid or not specified.

90 SIGNAL ENDPAGE statement executed.

91 ENVIRONMENT option invalid for file accessing indexed data set.

92 During opening of an index data set with the BKWD option, the attempt to position the data set at the last record failed.

93 Unidentified error detected by the operating system while opening a data set.

94 REUSE specified for a nonreusable data set.

95 Alternate index specified for an index data set is empty.

96 Incorrect OS/2 environment variable.

99 File cannot be opened.

150 SIGNAL STRINGSIZE statement executed or STRINGSIZE condition occurred.

151 Truncation occurred during assignment of a mixed character string.

290 SIGNAL INVALIDOP statement was executed or INVALIDOP exception occurred.

300 SIGNAL OVERFLOW statement executed or OVERFLOW condition occurred.

310 SIGNAL FIXEDOVERFLOW statement executed or FIXEDOVERFLOW condition occurred.

320 SIGNAL ZERODIVIDE statement executed or ZERODIVIDE condition occurred.

330 SIGNAL UNDERFLOW statement executed or UNDERFLOW condition occurred.

340 SIGNAL SIZE statement executed; or high-order nonzero digits have been lost in an assignment to a variable or temporary, or significant digits have been lost in an input/output operation.

341 High order nonzero digits have been lost in an input/output operation.

350 SIGNAL STRINGRANGE statement executed or STRINGRANGE condition occurred.

360 Attempt to allocate a based variable within an area that contains insufficient free storage for allocation to be made.

361 Insufficient space in target area for assignment of source area.

362 SIGNAL AREA statement executed.

400 SIGNAL ATTENTION statement executed.

450 SIGNAL STORAGE statement was executed.

451 ALLOCATE statement failed; insufficient storage to satisfy request.

500 SIGNAL CONDITION (name) statement executed.

520 SIGNAL SUBSCRIPTRANGE statement executed, or subscript has been evaluated and found to lie outside its specified bounds.

600 SIGNAL CONVERSION statement executed.

601 Invalid conversion attempted during input/output of a character string.

603 Error during processing of an F-format item for a GET STRING statement.

604 Error during processing of an F-format item for a GET FILE statement.

605 Error during processing of an F-format item for a GET FILE statement following a TRANSMIT condition.

606 Error during processing of an E-format item for a GET STRING statement.

607 Error during processing of an E-format item for a GET FILE statement.

608 Error during processing of an E-format item for a GET FILE statement following a TRANSMIT condition.

609 Error during processing of a B-format item for a GET STRING statement.

610 Error during processing of a B-format item for a GET FILE statement.

611 Error during processing of a B-format item for a GET FILE statement following TRANSMIT condition.

612 Error during character value to arithmetic conversion.

613 Error during character value to arithmetic conversion for a GET or PUT FILE statement.

614 Error during character value to arithmetic conversion for a GET or PUT FILE statement following a TRANSMIT condition.

615 Error during character value to bit value conversion.

616 Error during character value to bit value conversion for a GET or PUT FILE statement.

617 Error during character value to bit value conversion for a GET or PUT FILE statement following a TRANSMIT condition.

618 Error during character value to picture conversion.

619 Error during character value to picture conversion for a GET or PUT FILE statement.

620 Error during character value to picture conversion for a GET or PUT FILE statement following a TRANSMIT condition.

621 Error in decimal P-format item for a GET STRING statement.

622 Error in decimal P-format input for a GET FILE statement.

623 Error in decimal P-format input for a GET FILE statement following a TRANSMIT condition.

624 Error in character P-format input for a GET FILE statement.

625 Error exists in character P-format input for a GET FILE statement.

626 Error exists in character P-format input for a GET FILE statement following a TRANSMIT condition.

627 A graphic or mixed character string encountered in a nongraphic environment.

628 A graphic or mixed character string encountered in a nongraphic environment on input.

629 A graphic or mixed character string encountered in a nongraphic environment on input after TRANSMIT was detected.

633 An invalid character detected in a *X*, *BX*, or *GX* string constant.

634 An invalid character detected in a *X*, *BX*, or *GX* string constant on input.

635 An invalid character detected in a *X*, *BX*, or *GX* string constant on input after *TRANSMIT* was detected.

636 A shift character detected in a graphic string.

639 During processing of a mixed character string, one of the following occurred:

- A shift-in present in the SBCS portion.
- A shift-out present in the graphic (double-byte) portion. (A shift-out cannot appear in either byte of a graphic character).
- A shift-in present in the second byte of a graphic character.

640 Conversion from picture contained an invalid character.

641 Conversion from picture contained an invalid character on input or output.

642 Conversion from picture contained an invalid character on input after TRANSMIT was detected.

643 Error during processing of a graphic F-format item for a GET STRING statement.

644 Error during processing of a graphic F-format item for a GET FILE statement.

645 Error during processing of a graphic F-format item for a GET FILE statement following a TRANSMIT condition.

646 Error during processing of a graphic E-format item for a GET STRING statement.

647 Error during processing of a graphic E-format item for a GET FILE statement.

648 Error during processing of a graphic E-format item for a GET FILE statement following a TRANSMIT condition.

649 Error during processing of a graphic B-format item for a GET STRING statement.

650 Error during processing of a graphic B-format item for a GET FILE statement.

651 Error during processing of a graphic B-format item for a GET FILE statement following TRANSMIT condition.

652 Error during graphic character value to arithmetic conversion.

653 Error during graphic character value to arithmetic conversion for a GET or PUT FILE statement.

654 Error during graphic character value to arithmetic conversion for a GET or PUT FILE statement following a TRANSMIT condition.

655 Error during graphic character value to bit value conversion.

656 Error during graphic character value to bit value conversion for a GET or PUT FILE statement.

657 Error during graphic character value to bit value conversion for a GET or PUT FILE statement following a TRANSMIT condition.

658 Error during graphic character value to picture conversion.

659 Error during graphic character value to picture conversion for a GET or PUT FILE statement.

660 Error during graphic character value to picture conversion for a GET or PUT FILE statement following a TRANSMIT condition.

661 Error in decimal graphic P-format item for a GET STRING statement.

662 Error in decimal graphic P-format input for a GET FILE statement.

663 Error in decimal graphic P-format input for a GET FILE statement following a TRANSMIT condition.

664 Error in character graphic P-format input for a GET FILE statement.

665 Error exists in character graphic P-format input for a GET FILE statement.

666 Error exists in character graphic P-format input for a GET FILE statement following a TRANSMIT condition.

1002 GET or PUT STRING specifies data exceeding size of string.

1003 Further output prevented by TRANSMIT or KEY conditions previously raised for the data set.

1004 Attempt to use PAGE, LINE, or SKIP <= 0 for nonprint file.

1005 In a DISPLAY(expression) REPLY (character-reference) statement, expression or character-reference is zero length.

1007 A REWRITE or a DELETE statement not preceded by a READ.

1008 Unrecognized field preceding the assignment symbol in a string specified in a GET STRING DATA statement.

1009 An input/output statement specifies an operation or an option which conflicts with the file attributes.

1010 A built-in function or pseudovariable referenced an unopened file.

1011 Data management detected an input/output error but is unable to provide any information about its cause.

1013 Previous input operation incomplete; REWRITE or DELETE statement specifies data which has been previously read in by a READ statement with an EVENT option, and no corresponding WAIT has been executed.

1014 Attempt to initiate further input/output operation when number of incomplete operations equals number specified by ENVIRONMENT option NCP(n) or by default.

1015 Event variable specified for an input/output operation when already in use.

1016 After UNDEFINEDFILE condition raised as a result of an unsuccessful attempt to implicitly open a file, the file was found unopened on normal return from the ON-unit.

1018 End of file or string encountered in data before end of data-list or in edit-directed transmission format list.

1019 Attempt to close file not opened in current process.

1020 Further input/output attempted before WAIT statement executed to ensure completion of previous READ.

1021 Attempt to access a record locked by another file in this process.

1022 Unable to extend indexed data set.

1023 Exclusive file closed while records still locked in a subtask

1024 Incorrect sequence of I/O operations on device-associated file.

1025 Insufficient virtual storage available to complete request.

1026 No position established in index data set.

1027 Record control interval already held in exclusive control.

1028 Requested record lies on unmounted volume.

1029 Attempt to reposition in index data set failed.

1030 An error occurred during index upgrade on a index data set.

1031 Invalid sequential write attempted on index data set.

1040 A data set open for output used all available space.

1041 An attempt was made to write a record containing a record delimiter.

1042 Record in data set is not properly delimited.

1102 An error occurred in PL/I storage management. Storage to be freed was pointed to by an invalid address.

1104 An internal error occurred in PL/I library.

1105 Unable to create an object window.

1500 Computational error; short floating-point argument of SQRT built-in function is less than zero.

1501 Computational error; long floating-point argument of SQRT built-in function is less than zero.

1502 Computational error; extended floating-point argument of SQRT built-in function is less than zero.

1503 Computational error in LOG, LOG2, or LOG10 built-in function; extended floating-point argument is less than zero.

1504 Computational error in LOG, LOG2, or LOG10 built-in function; short floating-point argument is less than zero.

1505 Computational error in LOG, LOG2 or LOG10 built-in function; long floating-point argument is less than zero.

1506 Computational error in SIN, COS, SIND, or COSD built-in function; absolute value of short floating-point argument exceeds (2**63) (SIN and COS) or (2**63)*180 (SIND and COSD).

1507 Computational error in SIN, COS, SIND, or COSD built-in function; absolute value of long floating-point argument exceeds (2**63) (SIN and COS) or (2**63)*180 (SIND and COSD).

1508 Computational error; absolute value of short floating-point argument of TAN or TAND built-in function is greater than or equal to (2**63).

1509 Computational error; absolute value of long floating-point argument of TAN or TAND built-in function exceeds, respectively, (2**63) or (2**63)*180.

1510 Computational error; short floating-point arguments of ATAN or ATAND built-in function both invalid.

1511 Computational error; long floating-point arguments of ATAN or ATAND built-in function both invalid.

1514 Computational error; absolute value of short floating-point argument of ATANH built-in function >1.

1515 Computational error; absolute value of long floating-point argument of ATANH built-in function >1.

1516 Computational error; absolute value of extended floating-point argument of ATANH built-in function >1.

1517 Computational error in SIN, COS, SIND, or COSD built-in function; argument of extended floating-point argument exceeds (2**64).

1518 Computational error; absolute value of short floating-point argument of ASIN or ACOS built-in function exceeds 1.

1519 Computational error; absolute value of long floating-point argument of ASIN or ACOS built-in function exceeds 1.

1520 Computational error; absolute value of extended floating-point argument of ASIN, ACOS built-in function exceeds 1.

1521 Computational error; extended floating-point arguments of ATAN or ATAND built-in function both invalid.

1522 Computational error; absolute value of extended floating-point argument of TAN or TAND built-in function >= (2**64) or (2**64)*180, respectively.

1523 Computational error; absolute value of real short floating-point argument of SINH or COSH built-in function greater than 89.41.

1524 Absolute value of real long floating-point argument of SINH or COSH argument greater than or equal to 710.47.

1525 Absolute value of real extended floating-point argument of SINH or COSH than or equal to 11357.22.

1526 Computational error; absolute value of real short floating-point argument of COTAN or COTAND was greater than or equal to (2**63).

1527 Computational error; absolute value of real long floating-point argument of COTAN or COTAND was greater than or equal (2**63).

1528 Computational error; absolute value of real extended floating-point argument of COTAN or COTAND was greater than or equal to (2**64).

1529 Computational error in SIN, COS, SIND, or COSD built-in function; absolute value of the real part of complex short floating-point argument greater than or equal to (2**63)

1530 Computational error in SIN, COS, SIND, or COSD built-in function; absolute value of the real part of complex long floating-point argument greater than or equal to (2**63).

1531 Computational error in SIN, COS, SIND, or COSD built-in function; absolute value of the real part of complex extended floating-point argument was greater than or equal to (2**64).

1550 Computational error; during exponentiation, real short floating-point base is zero and integer exponent is not positive.

1551 Computational error; during exponentiation, real long floating-point base is zero and integer exponent is not positive.

1552 Computational error; during exponentiation, real short floating-point base is zero and the floating-point or noninteger exponent is not positive.

1553 Computational error; during exponentiation, real long floating-point base is zero and the floating-point or noninteger exponent is not positive.

1554 Computational error; during exponentiation, complex short floating-point base is zero and integer exponent is not positive.

1555 Computational error; during exponentiation, complex long floating-point base is zero and integer exponent is not positive.

1556 Computational error; during exponentiation, complex short floating-point base is zero and floating-point or noninteger exponent is not positive and real.

1557 Computational error; during exponentiation, complex long floating-point base is zero and floating-point or noninteger exponent is not positive and real.

1558 Computational error; complex short floating-point argument of ATAN or ATANH built-in function has value, respectively, of ±1I or ±1.

1559 Computational error; complex long floating-point argument of ATAN or ATANH built-in function has value, respectively, of ±1I or ±1.

1560 Computational error; during exponentiation, real extended floating-point base is zero and integer exponent not positive.

1561 Computational error; during exponentiation, real extended floating-point base is zero and floating-point or noninteger exponent is not positive.

1562 Computational error; during exponentiation, complex extended floating-point base is zero and integer exponent is not positive.

1563 Computational error; complex extended floating-point base is zero and floating-point or nonintegral exponent is not positive.

1564 Computational error; complex extended floating-point argument of ATAN or ATANH built-in function has value, respectively, of ±1I or ±1.

1565 Computational error; real short floating-point argument of EXP built-in function was less than –87.33.

1566 Computational error; real long floating-point argument of EXP built-in function was less than –708.39.

1567 Computational error; real extended floating-point argument of EXP built-in function was less than –11355.13.

1568 Computational error EXP built-in function; absolute value of the imaginary part of the complex short floating-point short argument greater than or equal to (2**63).

1569 Computational error EXP built-in function; absolute value of the imaginary part of the complex long floating-point short argument greater than or equal to (2**63).

1570 Computational error EXP built-in function; absolute value of the imaginary part of the complex extended floating-point short argument greater than or equal to (2**64).

1571 Computational error GAMMA or LOGGAMMA built-in function; real short floating point argument is greater than 35.04 (GAMMA) or 4.085E+36 (LOGGAMMA).

1572 Computational error GAMMA or LOGGAMMA built-in function; real long floating point argument is greater than 171.62 (GAMMA) or 2.559E+305 (LOGGAMMA).

1573 Computational error GAMMA or LOGGAMMA built-in function; real extended floating point argument is greater than 1755.54 (GAMMA) or 1.048E+4928 (LOGGAMMA).

1574 Computational error TANH built-in function; absolute value of the imaginary part of the complex short floating-point short argument greater than or equal to (2**63).

1575 Computational error TANH built-in function; absolute value of the imaginary part of the complex long floating-point short argument greater than or equal to (2**63).

1576 Computational error TANH built-in function; absolute value of the imaginary part of the complex extended floating-point short argument greater than or equal to (2**64).

1577 Computational error in LOG, LOG2, or LOG10 built-in function; real short floating-point argument equal to plus or minus zero.

1578 Computational error in LOG, LOG2, or LOG10 built-in function; real long floating-point argument equal to plus or minus zero.

1579 Computational error in LOG, LOG2, or LOG10 built-in function; real extended floating-point argument equal to plus zero.

1600 Computational error in EXP built-in function; for complex long floating-point arguments, the real argument was not plus or minus infinity, and the imaginary argument was not zero.

1601 Computational error in EXP built-in function; for complex extended floating-point arguments, the real argument was not plus or minus infinity, and the imaginary argument was not zero.

1602 Computational error; real part of the complex short floating-point argument for the EXP built-in function was not a valid IEEE number.

1603 Computational error; real part of the complex long floating-point argument for the EXP built-in function was not a valid IEEE number.

1604 Computational error; real part of the complex extended floating-point argument for the EXP built-in function was not a valid IEEE number.

1605 Computational error; imaginary part of the complex short floating-point argument for the EXP built-in function was not a valid IEEE number.

1606 Computational error; imaginary part of the complex long floating-point argument for the EXP built-in function was not a valid IEEE number.

1607 Computational error; imaginary part of the complex extended floating-point argument for the EXP built-in function was not a valid IEEE number.

1608 Computational error; both parts of the complex short floating-point argument for the EXP built-in function were not valid IEEE numbers.

1609 Computational error; both parts of the complex long floating-point argument for the EXP built-in function were not valid IEEE numbers.

1610 Computational error; both parts of the complex extended floating-point argument for the EXP built-in function were not valid IEEE numbers.

1611 Computational error; real short floating-point argument for EXP built-in function greater than or equal to 88.73.

1612 Computational error; real long floating-point argument for EXP built-in function greater than or equal to 709.79.

1613 Computational error; real extended floating-point argument for EXP built-in function greater than or equal to 11356.53.

1614 Computational error; real short floating-point argument for EXP built-in function is not a valid IEEE number.

1615 Computational error; real long floating-point argument for EXP built-in function is not a valid IEEE number.

1616 Computational error; real extended floating-point argument for EXP built-in function is not a valid IEEE number.

1617 Computational error in LOG built-in function; for complex short floating-point arguments, the real argument was not plus or minus infinity, and the imaginary argument was not zero.

1618 Computational error in LOG built-in function; for complex long floating-point arguments, the real argument was not plus or minus infinity, and the imaginary argument was not zero.

1619 Computational error in LOG, LOG2, or LOG10 built-in function; for complex extended floating-point arguments, the real argument was not plus or minus infinity, and the imaginary argument was not zero.

1620 Computational error in LOG, LOG2, or LOG10 built-in function; real part of complex short floating-point argument was not a valid IEEE number.

1621 Computational error in LOG, LOG2, or LOG10 built-in function; real part of complex long floating-point argument was not a valid IEEE number.

1622 Computational error in LOG, LOG2, or LOG10 built-in function; real part of complex extended floating-point argument was not a valid IEEE number.

1623 Computational error in LOG, LOG2, or LOG10 built-in function; imaginary part of complex short floating-point argument was not a valid IEEE number.

1624 Computational error in LOG, LOG2, or LOG10 built-in function; imaginary part of complex long floating-point argument was not a valid IEEE number.

1625 Computational error in LOG, LOG2, or LOG10 built-in function; imaginary part of complex extended floating-point argument was not a valid IEEE number.

1626 Computational error in LOG, LOG2, or LOG10 built-in function; both parts of complex short floating-point argument were not valid IEEE numbers.

1627 Computational error in LOG, LOG2, or LOG10 built-in function; both parts of complex long floating-point argument were not valid IEEE numbers.

1628 Computational error in LOG, LOG2, or LOG10 built-in function; both parts of complex extended floating-point argument were not valid IEEE numbers.

1629 Computational error in LOG, LOG2, or LOG10 built-in function; real short floating-point argument is not a valid IEEE number.

1630 Computational error in LOG, LOG2, or LOG10 built-in function; real long floating-point argument is not a valid IEEE number.

1631 Computational error in LOG, LOG2, or LOG10 built-in function; real extended floating-point argument is not a valid IEEE number.

1650 Computational error; during exponentiation, real long floating-point base is plus or minus infinity, and real long floating-point exponent is zero.

1651 Computational error; during exponentiation, real extended floating-point base is plus or minus infinity, and real extended floating-point exponent is zero.

1652 Computational error; during exponentiation for a real short floating-point base with a real short floating-point exponent, the first argument was not a valid IEEE number.

1653 Computational error; during exponentiation for a real long floating-point base with a real long floating-point exponent, the first argument was not a valid IEEE number.

1654 Computational error; during exponentiation for a real extended floating-point base with a real extended floating-point exponent, the first argument was not a valid IEEE number.

1655 Computational error; during exponentiation for a real short floating-point base with a real short floating-point exponent, the second argument was not a valid IEEE number.

1656 Computational error; during exponentiation for a real long floating-point base with a real long floating-point exponent, the second argument was not a valid IEEE number.

1657 Computational error; during exponentiation for a real extended floating-point base with a real extended floating-point exponent, the second argument was not a valid IEEE number.

1658 Computational error; during exponentiation for a real short floating-point base with a real short floating-point exponent, both arguments were not valid IEEE numbers.

1659 Computational error; during exponentiation for a real long floating-point base with a real long floating-point exponent both arguments were not valid IEEE numbers.

1660 Computational error; during exponentiation for a real extended floating-point base with a real extended floating-point exponent, both arguments were not valid IEEE numbers.

1661 Computational error; during exponentiation for complex short floating-point base with integer value exponent, an argument plus or minus infinity is specified.

1662 Computational error; during exponentiation for complex long floating-point base with integer value exponent, an argument plus or minus infinity is specified.

1663 Computational error; during exponentiation for complex extended floating-point base with integer value exponent, an argument plus or minus infinity is specified.

1664 Computational error; during exponentiation for complex short floating-point base with integer value exponent, the real part of the complex argument is not a valid IEEE number.

1665 Computational error; during exponentiation for complex long floating-point base with integer value exponent, the real part of the complex argument is not a valid IEEE number.

1666 Computational error; during exponentiation for complex extended floating-point base with integer value exponent, the real part of the complex argument is not a valid IEEE number.

1667 Computational error; during exponentiation for complex short floating-point base with integer value exponent, the imaginary part of the complex argument is not a valid IEEE number.

1668 Computational error; during exponentiation for complex long floating-point base with integer value exponent, the imaginary part of the complex argument is not a valid IEEE number.

1669 Computational error; during exponentiation for complex extended floating-point base with integer value exponent, the imaginary part of the complex argument is not a valid IEEE number.

1670 Computational error; during exponentiation for complex short floating-point base with integer value exponent, both parts of the complex argument are not valid IEEE numbers.

1671 Computational error; during exponentiation for complex long floating-point base with integer value exponent, both parts of the complex argument are not valid IEEE numbers.

1672 Computational error; during exponentiation for complex extended floating-point base with integer value exponent, both parts of the complex argument are not valid IEEE numbers.

1673 Computational error; during exponentiation, integer base is zero and integer exponent is not positive.

1674 Computational error; during exponentiation, integer base is not plus or minus 1 and integer exponent is not positive.

1675 Computational error; during exponentiation, real short floating-point base was plus or minus infinity and integer exponent is equal to plus or minus zero.

1676 Computational error; during exponentiation, real long floating-point base was plus or minus infinity and integer exponent is equal to plus or minus zero.

1677 Computational error; during exponentiation, real extended floating-point base was plus or minus infinity and integer exponent is equal to plus or minus zero.

1678 Computational error; during exponentiation for a real short floating-point base with an integer exponent, the first argument was not a valid IEEE number.

1679 Computational error; during exponentiation for a real long floating-point base with an integer exponent, the first argument was not a valid IEEE number.

1680 Computational error; during exponentiation for a real extended floating-point base with an integer exponent, the first argument was not a valid IEEE number.

1681 Computational error in the EXP built-in function; for complex short floating-point arguments, the real argument was not plus or minus infinity, and the imaginary argument was not zero.

1700 Computational error; during exponentiation for a complex long floating-point base with a complex long floating-point exponent, imaginary parts of both complex arguments are not valid IEEE numbers.

1701 Computational error; during exponentiation for a complex extended floating-point base with a complex extended floating-point exponent, imaginary parts of both complex arguments are not valid IEEE numbers.

1702 Computational error; during exponentiation for a complex short floating-point base with a complex short floating-point exponent, real part of first complex argument and imaginary part of second complex argument are not valid IEEE numbers.

1703 Computational error; during exponentiation for a complex long floating-point base with a complex long floating-point exponent, real part of first complex argument and imaginary part of second complex argument are not valid IEEE numbers.

1704 Computational error; during exponentiation for a complex extended floating-point base with a complex extended floating-point exponent, real part of first complex argument and imaginary part of second complex argument are not valid IEEE numbers.

1705 Computational error; during exponentiation for a complex short floating-point base with a complex short floating-point exponent, imaginary part of first complex argument and real part of second complex argument are not valid IEEE numbers.

1706 Computational error; during exponentiation for a complex long floating-point base with a complex long floating-point exponent, imaginary part of first complex argument and real part of second complex argument are not valid IEEE numbers.

1707 Computational error; during exponentiation for a complex extended floating-point base with a complex extended floating-point exponent, imaginary part of first complex argument and real part of second complex argument are not valid IEEE numbers.

1708 Computational error; during exponentiation for a complex short floating-point base with a complex short floating-point exponent, real part of first complex argument was the only valid IEEE number.

1709 Computational error; during exponentiation for a complex long floating-point base with a complex long floating-point exponent, real part of first complex argument was the only valid IEEE number.

1710 Computational error; during exponentiation for a complex extended floating-point base with a complex extended floating-point exponent, real part of first complex argument was the only valid IEEE number.

1711 Computational error; during exponentiation for a complex short floating-point base with a complex short floating-point exponent, imaginary part of first complex argument was the only valid IEEE number.

1712 Computational error; during exponentiation for a complex long floating-point base with a complex long floating-point exponent, imaginary part of first complex argument was the only valid IEEE number.

1713 Computational error; during exponentiation for a complex extended floating-point base with a complex extended floating-point exponent, imaginary part of first complex argument was the only valid IEEE number.

1714 Computational error; during exponentiation for a complex short floating-point base with a complex short floating-point exponent, real part of second complex argument was the only valid IEEE number.

1715 Computational error; during exponentiation for a complex long floating-point base with a complex long floating-point exponent, real part of second complex argument was the only valid IEEE number.

1716 Computational error; during exponentiation for a complex extended floating-point base with a complex extended floating-point exponent, real part of second complex argument was the only valid IEEE number.

1717 Computational error; during exponentiation for a complex short floating-point base with a complex short floating-point exponent, imaginary part of second complex argument was the only valid IEEE number.

1718 Computational error; during exponentiation for a complex long floating-point base with a complex long floating-point exponent, imaginary part of second complex argument was the only valid IEEE number.

1719 Computational error; during exponentiation for a complex extended floating-point base with a complex extended floating-point exponent, imaginary part of second complex argument was the only valid IEEE number.

1720 Computational error; during exponentiation for a complex short floating-point base with a complex short floating-point exponent, both parts of both complex arguments were not valid IEEE numbers.

1721 Computational error; during exponentiation for a complex long floating-point base with a complex long floating-point exponent, both parts of both complex arguments were not valid IEEE numbers.

1722 Computational error; during exponentiation for a complex extended floating-point base with a complex extended floating-point exponent, both parts of both complex arguments were not valid IEEE numbers.

1723 Computational error; during exponentiation, real short floating-point base plus or minus infinity and real short floating-point exponent is an invalid 32-bit integer.

1724 Computational error; during exponentiation, real long floating-point base is plus or minus infinity and real long floating-point exponent is an invalid 32-bit integer.

1725 Computational error; during exponentiation, real extended floating-point base plus or minus infinity and real extended floating-point exponent is an invalid 32-bit integer.

1726 Computational error; during exponentiation, real short floating-point base plus 1 and real short floating-point exponent is plus or minus infinity.

1727 Computational error; during exponentiation, real long floating-point base + 1 and real long floating-point exponent is plus or minus infinity.

1728 Computational error; during exponentiation, real extended floating-point base is + 1 and real extended floating-point exponent is plus or minus infinity.

1729 Computational error; during exponentiation, real short floating-point base is zero and real short floating-point exponent is not positive or zero.

1730 Computational error; during exponentiation, real long floating-point base is zero and real long floating-point exponent is not positive or zero.

1731 Computational error; during exponentiation, real short floating-point base plus or minus infinity and real short floating-point exponent is zero.

1750 Computational error; the first real short floating-point argument for SCALE was not a valid IEEE number.

1751 Computational error; the real short floating-point argument for ASIN(X) or ACOS(X) was not a valid IEEE number.

1752 Computational error; the real long floating-point argument for ASIN(X) or ACOS(X) was not a valid IEEE number.

1753 Computational error; the real extended floating-point argument for ASIN(X) or ACOS(X) was not a valid IEEE number.

1754 Computational error; during exponentiation for a complex short floating-point base with a complex short floating-point exponent, an argument exceeded the limit.

1755 Computational error; during exponentiation for a complex long floating-point base with a complex long floating-point exponent, an argument exceeded the limit.

1756 Computational error; during exponentiation for a complex extended floating-point base with a complex extended floating-point exponent, an argument exceeded the limit.

1757 Computational error; during exponentiation for a complex short floating-point base with a complex short floating-point exponent, plus or minus infinity was specified as an argument.

1758 Computational error; during exponentiation for a complex long floating-point base with a complex long floating-point exponent, plus or minus infinity was specified as an argument.

1759 Computational error; during exponentiation for a complex extended floating-point base with a complex extended floating-point exponent, plus or minus infinity was specified as an argument.

1760 Computational error; during exponentiation for a complex short floating-point base with a complex short floating-point exponent, the real part of the first complex argument is not a valid IEEE number.

1761 Computational error; during exponentiation for a complex long floating-point base with a complex long floating-point exponent, the real part of the first complex argument is not a valid IEEE number.

1762 Computational error; during exponentiation for a complex extended floating-point base with a complex extended floating-point exponent, the real part of the first complex argument is not a valid IEEE number.

1763 Computational error; during exponentiation for a complex short floating-point base with a complex short floating-point exponent, the real part of the second complex argument is not a valid IEEE number.

1764 Computational error; during exponentiation for a complex long floating-point base with a complex long floating-point exponent, the real part of the second complex argument is not a valid IEEE number.

1765 Computational error; during exponentiation for a complex extended floating-point base with a complex extended floating-point exponent, the real part of the second complex argument is not a valid IEEE number.

1766 Computational error; during exponentiation for a complex short floating-point base with a complex short floating-point exponent, the imaginary part of the first complex argument is not a valid IEEE number.

1767 Computational error; during exponentiation for a complex long floating-point base with a complex long floating-point exponent, the imaginary part of the first complex argument is not a valid IEEE number.

1768 Computational error; during exponentiation for a complex extended floating-point base with a complex extended floating-point exponent, the imaginary part of the first complex argument is not a valid IEEE number.

1769 Computational error; during exponentiation for a complex short floating-point base with a complex short floating-point exponent, the imaginary part of the second complex argument is not a valid IEEE number.

1770 Computational error; during exponentiation for a complex long floating-point base with a complex long floating-point exponent, the imaginary part of the second complex argument is not a valid IEEE number.

1771 Computational error; during exponentiation for a complex extended floating-point base with a complex extended floating-point exponent, the imaginary part of the second complex argument is not a valid IEEE number.

1772 Computational error; during exponentiation for a complex short floating-point base with a complex short floating-point exponent, both parts of the first complex argument are not valid IEEE numbers.

1773 Computational error; during exponentiation for a complex long floating-point base with a complex long floating-point exponent, both parts of the first complex argument are not valid IEEE numbers.

1774 Computational error; during exponentiation for a complex extended floating-point base with a complex extended floating-point exponent, both parts of the first complex argument are not valid IEEE numbers.

1775 Computational error; during exponentiation for a complex short floating-point base with a complex short floating-point exponent, both parts of the second complex argument are not valid IEEE numbers.

1776 Computational error; during exponentiation for a complex long floating-point base with a complex long floating-point exponent, both parts of the second complex argument are not valid IEEE numbers.

1777 Computational error; during exponentiation for a complex extended floating-point base with a complex extended floating-point exponent, both parts of the second complex argument are not valid IEEE numbers.

1778 Computational error; during exponentiation for a complex short floating-point base with a complex short floating-point exponent, real parts of both complex arguments are not valid IEEE numbers.

1779 Computational error; during exponentiation for a complex long floating-point base with a complex long floating-point exponent, real parts of both complex arguments are not valid IEEE numbers.

1780 Computational error; during exponentiation for a complex extended floating-point base with a complex extended floating-point exponent, real parts of both complex arguments are not valid IEEE numbers.

1781 Computational error; during exponentiation for a complex short floating-point base with a complex short floating-point exponent, imaginary parts of both complex arguments are not valid IEEE numbers.

1800 Computational error in SIN, COS, SIND, or COSD built-in function; for complex extended floating-point argument both parts of the argument was not a valid IEEE number.

1801 Computational error in SIN, COS, SIND, or COSD built-in function; absolute value of real short floating-point argument is not a valid IEEE number.

1802 Computational error in SIN, COS, SIND, or COSD built-in function; absolute value of real long floating-point argument is not a valid IEEE number.

1803 Computational error in SIN, COS, SIND, or COSD built-in function; absolute value of real extended floating-point argument is not a valid IEEE number.

1804 The calculated result of real extended floating-point arguments for TANH overflowed the output field.

1808 Computational error; for real short floating-point arguments of ATAN or ATAND built-in function, the first argument was not a valid IEEE number.

1809 Computational error; for real long floating-point arguments of ATAN or ATAND built-in function, the first argument was not a valid IEEE number.

1810 Computational error; for real extended floating-point argument of ATAN or ATAND built-in function, the first argument was not a valid IEEE number.

1811 Computational error; for real short floating-point arguments of ATAN or ATAND built-in function, the second argument was not a valid IEEE number.

1812 Computational error; for real long floating-point arguments of ATAN or ATAND built-in function, the second argument was not a valid IEEE number.

1813 Computational error; for real extended floating-point argument of ATAN or ATAND built-in function, the second argument was not a valid IEEE number.

1814 Computational error; both real short floating-point arguments of ATAN or ATAND built-in function were not valid IEEE numbers.

1815 Computational error; both real long floating-point arguments of ATAN or ATAND built-in function were not valid IEEE numbers.

1816 Computational error; both real extended floating-point arguments of ATAN or ATAND built-in function were not valid IEEE numbers.

1817 Computational error; complex short floating-point argument of ATAN or ATAND built-in function does not have value of (plus infinity, 0i) or (minus infinity, 0i).

1818 Computational error; complex long floating-point argument of ATAN or ATAND built-in function does not have value of (plus infinity, 0i) or (minus infinity, 0i).

1819 Computational error; complex extended floating-point argument of ATAN or ATAND built-in function does not have value of (plus infinity, 0i) or (minus infinity, 0i).

1820 Computational error; real part of complex short floating-point argument of ATAN or ATAND built-in function is not a valid IEEE number.

1821 Computational error; real part of complex long floating-point argument of ATAN or ATAND built-in function is not a valid IEEE number.

1822 Computational error; real part of complex extended floating-point argument of ATAN or ATAND built-in function is not a valid IEEE number.

1823 Computational error; imaginary part of complex short floating-point argument of ATAN or ATAND built-in function is not a valid IEEE number.

1824 Computational error; imaginary part of complex long floating-point argument of ATAN or ATAND built-in function is not a valid IEEE number.

1825 Computational error; imaginary part of complex extended floating-point argument of ATAN or ATAND built-in function is not a valid IEEE number.

1826 Computational error; both parts of complex short floating-point argument of ATAN or ATAND built-in function were not valid IEEE numbers.

1827 Computational error; both parts of complex long floating-point argument of ATAN or ATAND built-in function were not valid IEEE numbers.

1828 Computational error; both parts of complex extended floating-point argument of ATAN or ATAND built-in function were not valid IEEE numbers.

1829 Computational error; the real short floating-point argument of ATAN(X) or ATAND(X) built-in function was not a valid IEEE number.

1830 Computational error; the real long floating-point argument of ATAN(X) or ATAND(X) built-in function was not a valid IEEE number.

1831 Computational error; the real extended floating-point argument of ATAN(X) or ATAND(X) built-in function was not a valid IEEE number.

1850 Computational error; real short floating-point argument of COTAN or COTAND was not a valid IEEE number.

1851 Computational error; real long floating-point argument of COTAN or COTAND was not a valid IEEE number.

1852 Computational error; real extended floating-point argument of COTAN or COTAND was not a valid IEEE number.

1853 Computational error in TAN or TAND; for complex short floating-point argument, absolute value of the real part of argument greater than or equal to (2**63).

1854 Computational error in TAN or TAND; for complex long floating-point argument, absolute value of the real part of argument greater than or equal to (2**63).

1855 Computational error in TAN or TAND; for complex extended floating-point argument, absolute value of the real part of argument greater than or equal to (2**64).

1856 Computational error in TAN or TAND; for complex short floating-point argument both parts of the argument were plus or minus infinity.

1857 Computational error in TAN or TAND; for complex long floating-point argument both parts of the argument were plus or minus infinity.

1858 Computational error in TAN or TAND; for complex extended floating-point argument both parts of the argument were plus or minus infinity.

1859 Computational error in TAN or TAND; for complex short floating-point argument real part of argument not a valid IEEE number.

1860 Computational error in TAN or TAND; for complex long floating-point argument real part of argument not a valid IEEE number.

1861 Computational error in TAN or TAND; for complex extended floating-point argument real part of argument not a valid IEEE number.

1862 Computational error in TAN or TAND; for complex short floating-point argument imaginary part of argument not a valid IEEE number.

1863 Computational error in TAN or TAND; for complex long floating-point argument imaginary part of argument not a valid IEEE number.

1864 Computational error in TAN or TAND; for complex extended floating-point argument imaginary part of argument not a valid IEEE number.

1865 Computational error in TAN or TAND; for complex short floating-point argument both parts of the argument were not valid IEEE numbers.

1866 Computational error in TAN or TAND; for complex long floating-point argument both parts of the argument were not valid IEEE numbers.

1867 Computational error in TAN or TAND; for complex extended floating-point argument both parts of the argument were not valid IEEE numbers.

1868 Computational error in TAN or TAND; real short floating-point argument not a valid IEEE number.

1869 Computational error in TAN or TAND; real long floating-point argument not a valid IEEE number.

1870 Computational error in TAN or TAND; real extended floating-point argument not a valid IEEE number.

1871 Computational error in SIN, COS, SIND, or COSD built-in function; for complex short floating-point argument both parts of the argument were plus or minus infinity.

1872 Computational error in SIN, COS, SIND, or COSD built-in function; for complex long floating-point argument both parts of the argument were plus or minus infinity.

1873 Computational error in SIN, COS, SIND, or COSD built-in function; for complex extended floating-point argument both parts of the argument were plus or minus infinity.

1874 Computational error in SIN, COS, SIND, or COSD built-in function; for complex short floating-point argument the real part of the argument was not a valid IEEE number.

1875 Computational error in SIN, COS, SIND, or COSD built-in function; for complex long floating-point argument the real part of the argument was not a valid IEEE number.

1876 Computational error in SIN, COS, SIND, or COSD built-in function; for complex extended floating-point argument the real part of the argument was not a valid IEEE number.

1877 Computational error in SIN, COS, SIND, or COSD built-in function; for complex short floating-point argument the imaginary part of the argument was not a valid IEEE number.

1878 Computational error in SIN, COS, SIND, or COSD built-in function; for complex long floating-point argument the imaginary part of the argument was not a valid IEEE number.

1879 Computational error in SIN, COS, SIND, or COSD built-in function; for complex extended floating-point argument the imaginary part of the argument was not a valid IEEE number.

1880 Computational error in SIN, COS, SIND, or COSD built-in function; for complex short floating-point argument both parts of the argument were not valid IEEE numbers.

1881 Computational error in SIN, COS, SIND, or COSD built-in function; for complex long floating-point argument both parts of the argument were not valid IEEE numbers.

1900 Computational error in TANH; for complex long floating-point argument the real part of the argument was not equal to plus or minus infinity, and the imaginary part of the argument was not zero.

1901 Computational error in TANH; for complex extended floating-point argument the real part of the argument was not equal to plus or minus infinity, and the imaginary part of the argument was not zero.

1902 Computational error in TANH; for complex short floating-point argument real part of argument not a valid IEEE number.

1903 Computational error in TANH; for complex long floating-point argument real part of argument not a valid IEEE number.

1904 Computational error in TANH; for complex extended floating-point argument real part of argument not a valid IEEE number.

1905 Computational error in TANH; for complex short floating-point argument the imaginary part of the argument was not a valid IEEE number.

1906 Computational error in TANH; for complex long floating-point argument the imaginary part of the argument was not a valid IEEE number.

1907 Computational error in TANH; for complex extended floating-point argument the imaginary part of the argument was not a valid IEEE number.

1908 Computational error in TANH; for complex short floating-point argument both parts of the argument were not valid IEEE numbers.

1909 Computational error in TANH; for complex long floating-point argument both parts of the argument were not valid IEEE numbers.

1910 Computational error in TANH; for complex extended floating-point argument both parts of the argument were not valid IEEE numbers.

1911 Computational error; real short floating-point argument of TANH built-in function not a valid IEEE number.

1912 Computational error; real long floating-point argument of TANH built-in function not a valid IEEE number.

1913 Computational error; real extended floating-point argument of TANH built-in function not a valid IEEE number.

1914 Computational error; absolute value of imaginary part of complex short floating-point argument of SINH or COSH built-in function was greater than or equal to (2**63).

1915 Computational error; absolute value of the imaginary part of complex long floating-point argument of SINH or COSH built-in function was greater than or equal to (2**63).

1916 Computational error; absolute value of the imaginary part of complex extended floating-point argument of SINH or COSH built-in function was greater than or equal to (2**64).

1917 Computational error; for complex short floating-point argument of SINH or COSH built-in function real argument was not plus or minus infinity and imaginary argument was not zero.

1918 Computational error; for complex long floating-point argument of SINH or COSH built-in function real argument was not plus or minus infinity and imaginary argument was not zero.

1919 Computational error; for complex extended floating-point argument of SINH or COSH built-in function real argument was not plus or minus infinity and imaginary argument was not zero.

1920 Computational error; for complex short floating-point argument of SINH or COSH built-in function real part of argument not valid IEEE number.

1921 Computational error; for complex long floating-point argument of SINH or COSH built-in function real part of argument not valid IEEE number.

1922 Computational error; for complex extended floating-point argument of SINH or COSH built-in function real part of argument not valid IEEE number.

1923 Computational error; for complex short floating-point argument of SINH or COSH built-in function imaginary part of argument not valid IEEE number.

1924 Computational error; for complex long floating-point argument of SINH or COSH built-in function imaginary part of argument not valid IEEE number.

1925 Computational error; for complex extended floating-point argument of SINH or COSH built-in function imaginary part of argument not valid IEEE number.

1926 Computational error; for complex short floating-point argument of SINH or COSH built-in function both parts of argument not valid IEEE numbers.

1927 Computational error; for complex long floating-point argument of SINH or COSH built-in function both parts of argument not valid IEEE numbers.

1928 Computational error; for complex extended floating-point argument of SINH or COSH built-in function both parts of argument not valid IEEE numbers.

1929 Computational error; real short floating-point argument of SINH or COSH built-in function was not a valid IEEE number.

1930 Computational error; real long floating-point argument of SINH or COSH built-in function was not a valid IEEE number.

1931 Computational error; real extended floating-point argument of SINH or COSH built-in function was not a valid IEEE number.

1950 Computational error in SQRT; for complex extended floating-point argument real part was not equal to plus or minus infinity, and imaginary part was not equal to zero.

1951 Computational error in SQRT; real part of complex short floating-point argument was not a valid IEEE number.

1952 Computational error in SQRT; real part of complex long floating-point argument was not a valid IEEE number.

1953 Computational error in SQRT; real part of complex extended floating-point argument was not a valid IEEE number.

1954 Computational error in SQRT; imaginary part of complex short floating-point argument was not a valid IEEE number.

1955 Computational error in SQRT; imaginary part of complex long floating-point argument was not a valid IEEE number.

1956 Computational error in SQRT; imaginary part of complex extended floating-point argument was not a valid IEEE number.

1957 Computational error in SQRT; both parts of complex short floating-point argument were not valid IEEE numbers.

1958 Computational error in SQRT; both parts of complex long floating-point argument were not valid IEEE numbers.

1959 Computational error in SQRT; both parts of complex extended floating-point argument were not valid IEEE numbers.

1960 Computational error in SQRT; real short floating-point argument is equal to minus zero.

1961 Computational error in SQRT; real long floating-point argument is equal to minus zero.

1962 Computational error in SQRT; real extended floating-point argument is equal to minus zero.

1963 Computational error in SQRT; real short floating-point argument was not a valid IEEE number.

1964 Computational error in SQRT; real long floating-point argument was not a valid IEEE number.

1965 Computational error in SQRT; real extended floating-point argument was not a valid IEEE number.

1966 Computational error; complex short floating-point argument of ATANH included plus or minus infinity.

1967 Computational error; complex long floating-point argument of ATANH included plus or minus infinity.

1968 Computational error; complex extended floating-point argument of ATANH included plus or minus infinity.

1969 Computational error; real part of complex short floating-point argument of ATANH was not a valid IEEE number.

1970 Computational error; real part of complex long floating-point argument of ATANH was not a valid IEEE number.

1971 Computational error; real part of complex extended floating-point argument of ATANH was not a valid IEEE number.

1972 Computational error; imaginary part of complex short floating-point argument of ATANH was not a valid IEEE number.

1973 Computational error; imaginary part of complex long floating-point argument of ATANH was not a valid IEEE number.

1974 Computational error; imaginary part of complex extended floating-point argument of ATANH was not a valid IEEE number.

1975 Computational error; both parts of complex short floating-point argument of ATANH were not valid IEEE numbers.

1976 Computational error; both parts of complex long floating-point argument of ATANH were not valid IEEE numbers.

1977 Computational error; both parts of complex extended floating-point argument of ATANH were not valid IEEE numbers.

1978 Computational error; floating-point argument of ATANH was not a valid IEEE number.

1979 Computational error; long floating-point argument of ATANH was not a valid IEEE number.

1980 Computational error; of extended floating-point argument of ATANH was not a valid IEEE number.

1981 Computational error in TANH; for complex short floating-point argument the real part of the argument was not equal to plus or minus infinity, and the imaginary part of the argument was not zero.

2002 WAIT statement cannot be executed because of restricted system facility.

2101 Greenwich mean time was not available for the RANDOM built-in function.

2102 An invalid seed value was detected in the RANDOM built-in function. The random number was set to -1.

2103 Local time was unavailable.

2150 Computational error; in MOD(x,y) built-in function the second argument was equal to zero.

2151 Computational error in ABS built-in function; real part of complex short floating-point argument was not a valid IEEE number.

2152 Computational error in ABS built-in function; real part of complex long floating-point argument was not a valid IEEE number.

2153 Computational error in ABS built-in function; real part of complex extended floating-point argument was not a valid IEEE number.

2154 Computational error in ABS built-in function; imaginary part of complex short floating-point argument was not a valid IEEE number.

2155 Computational error in ABS built-in function; imaginary part of complex long floating-point argument was not a valid IEEE number.

2156 Computational error in ABS built-in function; imaginary part of complex extended floating-point argument was not a valid IEEE number.

2157 Computational error in ABS built-in function; both parts of complex short floating-point argument were not valid IEEE numbers.

2158 Computational error in ABS built-in function; both parts of complex long floating-point argument were not valid IEEE numbers.

2159 Computational error in ABS built-in function; both parts of complex extended floating-point argument were not valid IEEE numbers.

2160 Computational error in ABS built-in function; integer argument is equal to (–2**31).

2161 Computational error in ABS built-in function; real short floating-point argument was not a valid IEEE number.

2162 Computational error in ABS built-in function; real long floating-point argument was not a valid IEEE number.

2163 Computational error in ABS built-in function; real extended floating-point argument was not a valid IEEE number.

2164 Computational error GAMMA or LOGGAMMA built-in function; real extended floating point argument is less than zero.

2165 Computational error GAMMA or LOGGAMMA built-in function; real short floating point argument is less than or equal to zero.

2166 Computational error GAMMA or LOGGAMMA built-in function; real long floating point argument is less than or equal to zero.

2167 Computational error GAMMA or LOGGAMMA built-in function; real extended floating point argument is equal to zero.

2168 Computational error GAMMA or LOGGAMMA built-in function; real short floating point argument is not a valid IEEE number.

2169 Computational error GAMMA or LOGGAMMA built-in function; real long floating point argument is not a valid IEEE number.

2170 Computational error GAMMA or LOGGAMMA built-in function; real extended floating point argument is not a valid IEEE number.

2171 Computational error in ERFC built-in function; real short floating-point argument was greater than 9.19.

2172 Computational error in ERFC built-in function; real long floating-point argument was greater than 26.54.

2173 Computational error in ERFC built-in function; real extended floating-point argument was greater than 106.53.

2174 Computational error in ERFC built-in function; real short floating-point argument was not a valid IEEE number.

2175 Computational error in ERFC built-in function; real long floating-point argument was not a valid IEEE number.

2176 Computational error in ERFC built-in function; real extended floating-point argument was not a valid IEEE number.

2177 Real short floating-point argument in ERF was not a valid IEEE number.

2178 Real long floating-point argument in ERF was not a valid IEEE number.

2179 Real extended floating-point argument in ERF was not a valid IEEE number.

2180 Computational error in SQRT; for complex short floating-point argument, real part was not equal to plus or minus infinity, and imaginary part was not equal to zero.

2181 Computational error in SQRT; for complex long floating-point argument, real part was not equal to plus or minus infinity, and imaginary part was not equal to zero.

2200 Computational error; during multiplication real part of first complex long floating-point argument was the only valid IEEE number.

2201 Computational error; during multiplication real part of first complex extended floating-point argument was the only valid IEEE number.

2202 Computational error; during multiplication the imaginary part of the first complex short floating-point argument was the only valid IEEE number.

2203 Computational error; during multiplication the imaginary part of the first complex long floating-point argument was the only valid IEEE number.

2204 Computational error; during multiplication the imaginary part of the first complex extended floating-point argument was the only valid IEEE number.

2205 Computational error; during multiplication the real part of the second complex short floating-point argument was the only valid IEEE number.

2206 Computational error; during multiplication the real part of the second complex long floating-point argument was the only valid IEEE number.

2207 Computational error; during multiplication the real part of the second complex extended floating-point argument was the only valid IEEE number.

2208 Computational error; during multiplication the imaginary part of the second complex short floating-point argument was the only valid IEEE number.

2209 Computational error; during multiplication the imaginary part of the second complex long floating-point argument was the only valid IEEE number.

2210 Computational error; during multiplication the imaginary part of the second complex extended floating-point argument was the only valid IEEE number.

2211 Computational error; during multiplication both parts of both complex short floating-point arguments were not valid IEEE numbers.

2212 Computational error; during multiplication both parts of both complex long floating-point arguments were not valid IEEE numbers.

2213 Computational error; during multiplication both parts of both complex extended floating-point arguments were not valid IEEE numbers.

2214 The real short floating-point argument for TRUNC was plus or minus infinity.

2215 The real long floating-point argument for TRUNC was plus or minus infinity.

2216 The real extended floating-point argument for TRUNC was plus or minus infinity.

2217 The real short floating-point argument for TRUNC was not a valid IEEE number.

2218 The real long floating-point argument for TRUNC was not a valid IEEE number.

2219 The real extended floating-point argument for TRUNC was not a valid IEEE number.

2220 Computational error; in MOD(x,y) built-in function real short floating-point arguments, the first argument was plus or minus infinity, or the second argument was plus or minus zero.

2221 Computational error; in MOD(x,y) built-in function real long floating-point arguments, the first argument was plus or minus infinity, or the second argument was plus or minus zero.

2222 Computational error; in MOD(x,y) built-in function real extended floating-point arguments, the first argument was plus or minus infinity, or the second argument was plus or minus zero.

2223 Computational error; in MOD(x,y) built-in function real short floating-point arguments, the first argument was not a valid IEEE number.

2224 Computational error; in MOD(x,y) built-in function real long floating-point arguments, the first argument was not a valid IEEE number.

2225 Computational error; in MOD(x,y) built-in function real extended floating-point arguments, the first argument was not a valid IEEE number.

2226 Computational error; in MOD(x,y) built-in function real short floating-point arguments, the second argument was not a valid IEEE number.

2227 Computational error; in MOD(x,y) built-in function real long floating-point arguments, the second argument was not a valid IEEE number.

2228 Computational error; in MOD(x,y) built-in function real extended floating-point arguments, the second argument was not a valid IEEE number.

2229 Computational error; in MOD(x,y) built-in function real short floating-point arguments, both arguments were not valid IEEE numbers

2230 Computational error; in MOD(x,y) built-in function real long floating-point arguments, both arguments were not valid IEEE numbers

2231 Computational error; in MOD(x,y) bulit-in function real extended floating-point arguments, both arguments were not valid IEEE numbers.

2250 Computational error; during multiplication for complex extended floating-point arguments plus or minus infinity was specified.

2251 Computational error; during multiplication the real part of the first complex short floating-point argument was not a valid IEEE number.

2252 Computational error; during multiplication the real part of the first complex long floating-point argument was not a valid IEEE number.

2253 Computational error; during multiplication the real part of the first complex extended floating-point argument was not a valid IEEE number.

2254 Computational error; during multiplication the real part of the second complex short floating-point argument was not a valid IEEE number.

2255 Computational error; during multiplication the real part of the second complex long floating-point argument was not a valid IEEE number.

2256 Computational error; during multiplication the real part of the second complex extended floating-point argument was not a valid IEEE number.

2257 Computational error; during multiplication the imaginary part of the first complex short floating-point argument was not a valid IEEE number.

2258 Computational error; during multiplication the imaginary part of the first complex long floating-point argument was not a valid IEEE number.

2259 Computational error; during multiplication the imaginary part of the first complex extended floating-point argument was not a valid IEEE number.

2260 Computational error; during multiplication the imaginary part of the second complex short floating-point argument was not a valid IEEE number.

2261 Computational error; during multiplication the imaginary part of the second complex long floating-point argument was not a valid IEEE number.

2262 Computational error; during multiplication the imaginary part of the second complex extended floating-point argument was not a valid IEEE number.

2263 Computational error; during multiplication both parts of first complex short floating-point arguments were not valid IEEE numbers.

2264 Computational error; during multiplication both parts of first complex long floating-point arguments were not valid IEEE numbers.

2265 Computational error; during multiplication both parts of first complex extended floating-point arguments were not valid IEEE numbers.

2266 Computational error; during multiplication both parts of second complex short floating-point arguments were not valid IEEE numbers.

2267 Computational error; during multiplication both parts of second complex long floating-point arguments were not valid IEEE numbers.

2268 Computational error; during multiplication both parts of second complex extended floating-point arguments were not valid IEEE numbers.

2269 Computational error; during multiplication real parts of both complex short floating-point arguments were not valid IEEE numbers.

2270 Computational error; during multiplication real parts of both complex long floating-point arguments were not valid IEEE numbers.

2271 Computational error; during multiplication real parts of both complex extended floating-point arguments were not valid IEEE numbers.

2272 Computational error; during multiplication imaginary parts of both complex short floating-point arguments were not valid IEEE numbers.

2273 Computational error; during multiplication imaginary parts of both complex long floating-point arguments were not valid IEEE numbers.

2274 Computational error; during multiplication imaginary parts of both complex extended floating-point arguments were not valid IEEE numbers.

2275 Computational error; during multiplication real part of first complex short floating-point argument and imaginary part of second complex short floating-point argument were not valid IEEE numbers.

2276 Computational error; during multiplication real part of first complex long floating-point argument and imaginary part of second complex long floating-point argument were not valid IEEE numbers.

2277 Computational error; during multiplication real part of first complex extended floating-point argument and imaginary part of second complex extended floating-point argument were not valid IEEE numbers.

2278 Computational error; during multiplication imaginary part of first complex short floating-point argument and real part of second complex short floating-point argument were not valid IEEE numbers.

2279 Computational error; during multiplication imaginary part of first complex long floating-point argument and real part of second complex long floating-point argument were not valid IEEE numbers.

2280 Computational error; during multiplication imaginary part of first complex extended floating-point argument and real part of second complex extended floating-point argument were not valid IEEE numbers.

2281 Computational error; during multiplication real part of first complex short floating-point argument was the only valid IEEE number.

2300 Computational error; during division real parts of both complex short floating-point arguments were not valid IEEE numbers.

2301 Computational error; during division real parts of both complex long floating-point arguments were not valid IEEE numbers.

2302 Computational error; during division real parts of both complex extended floating-point arguments were not valid IEEE numbers.

2303 Computational error; during division imaginary parts of both complex short floating-point arguments were not valid IEEE numbers.

2304 Computational error; during division imaginary parts of both complex long floating-point arguments were not valid IEEE numbers.

2305 Computational error; during division imaginary parts of both complex extended floating-point arguments were not valid IEEE numbers.

2306 Computational error; during division real part of first complex short floating-point argument and imaginary part of second complex short floating-point argument were not valid IEEE numbers.

2307 Computational error; during division real part of first complex long floating-point argument and imaginary part of second complex long floating-point argument were not valid IEEE numbers.

2308 Computational error; during division real part of first complex extended floating-point argument and imaginary part of second complex extended floating-point argument were not valid IEEE numbers.

2309 Computational error; during division imaginary part of first complex short floating-point argument and real part of second complex short floating-point argument were not valid IEEE numbers.

2310 Computational error; during division imaginary part of first complex long floating-point argument and real part of second complex long floating-point argument were not valid IEEE numbers.

2311 Computational error; during division imaginary part of first complex extended floating-point argument and real part of second complex extended floating-point argument were not valid IEEE numbers.

2312 Computational error; during division real part of first complex short floating-point argument was the only valid IEEE number.

2313 Computational error; during division real part of first complex long floating-point argument was the only valid IEEE number.

2314 Computational error; during division real part of first complex extended floating-point argument was the only valid IEEE number.

2315 Computational error; during division imaginary part of first complex short floating-point argument was the only valid IEEE number.

2316 Computational error; during division imaginary part of first complex long floating-point argument was the only valid IEEE number.

2317 Computational error; during division imaginary part of first complex extended floating-point argument was the only valid IEEE number.

2318 Computational error; during division real part of second complex short floating-point argument was the only valid IEEE number.

2319 Computational error; during division real part of second complex long floating-point argument was the only valid IEEE number.

2320 Computational error; during division real part of second complex extended floating-point argument was the only valid IEEE number.

2321 Computational error; during division imaginary part of second complex short floating-point argument was the only valid IEEE number.

2322 Computational error; during division imaginary part of second complex long floating-point argument was the only valid IEEE number.

2323 Computational error; during division imaginary part of second complex extended floating-point argument was the only valid IEEE number.

2324 Computational error; during division both parts of both complex short floating-point argument were not valid IEEE numbers.

2325 Computational error; during division both parts of both complex long floating-point argument were not valid IEEE numbers.

2326 Computational error; during division both parts of both complex extended floating-point argument were not valid IEEE numbers.

2327 Computational error; during multiplication complex short floating-point arguments equal to the limits.

2328 Computational error; during multiplication complex long floating-point arguments equal to the limits.

2329 Computational error; during multiplication complex extended floating-point arguments equal to the limits.

2330 Computational error; during multiplication for complex short floating-point arguments plus or minus infinity was specified.

2331 Computational error; during multiplication for complex long floating-point arguments plus or minus infinity was specified.

2350 Computational error; the first real long floating-point argument for SCALE was not a valid IEEE number.

2351 Computational error; the first real extended floating-point argument for SCALE was not a valid IEEE number.

2352 X in CEIL(X) or FLOOR(X) was invalid for a real short floating-point argument because the argument was plus or minus infinity.

2353 X in CEIL(X) or FLOOR(X) was invalid for a real long floating-point argument because the argument was plus or minus infinity.

2354 X in CEIL(X) or FLOOR(X) was invalid for a real extended floating-point argument because the argument was plus or minus infinity.

2355 X in CEIL(X) or FLOOR(X) was invalid for a real short floating-point argument because the argument was not a valid IEEE number.

2356 X in CEIL(X) or FLOOR(X) was invalid for a real long floating-point argument because the argument was not a valid IEEE number.

2357 X in CEIL(X) or FLOOR(X) was invalid for a real extended floating-point argument because the argument was not a valid IEEE number.

2358 Computational error; during division complex short floating-point arguments equal to the limits.

2359 Computational error; during division complex long floating-point arguments equal to the limits.

2360 Computational error; during division complex extended floating-point arguments equal to the limits.

2361 Computational error; during division for complex short floating-point arguments plus or minus infinity was specified.

2362 Computational error; during division for complex long floating-point arguments plus or minus infinity was specified.

2363 Computational error; during division for complex extended floating-point arguments plus or minus infinity was specified.

2364 Computational error; during division real part of first complex short floating-point argument was not a valid IEEE number.

2365 Computational error; during division real part of first complex long floating-point argument was not a valid IEEE number.

2366 Computational error; during division real part of first complex extended floating-point argument was not a valid IEEE number.

2367 Computational error; during division real part of second complex short floating-point argument was not a valid IEEE number.

2368 Computational error; during division real part of second complex long floating-point argument was not a valid IEEE number.

2369 Computational error; during division real part of second complex extended floating-point argument was not a valid IEEE number.

2370 Computational error; during division imaginary part of first complex short floating-point argument was not a valid IEEE number.

2371 Computational error; during division imaginary part of first complex long floating-point argument was not a valid IEEE number.

2372 Computational error; during division imaginary part of first complex extended floating-point argument was not a valid IEEE number.

2373 Computational error; during division imaginary part of second complex short floating-point argument was not a valid IEEE number.

2374 Computational error; during division imaginary part of second complex long floating-point argument was not a valid IEEE number.

2375 Computational error; during division imaginary part of second complex extended floating-point argument was not a valid IEEE number.

2376 Computational error; during division both parts of first complex short floating-point argument were not valid IEEE numbers.

2377 Computational error; during division both parts of first complex long floating-point argument were not valid IEEE numbers.

2378 Computational error; during division both parts of first complex extended floating-point argument were not valid IEEE numbers.

2379 Computational error; during division both parts of second complex short floating-point argument were not valid IEEE numbers.

2380 Computational error; during division both parts of second complex long floating-point argument were not valid IEEE numbers.

2381 Computational error; during division both parts of second complex extended floating-point argument were not valid IEEE numbers.

2403 Computational error; real extended floating point argument of GAMMA or LOGAMMA built-in function was less than or equal to minus zero.

2404 Computational error; real extended floating point argument of GAMMA or LOGAMMA built-in function was equal to zero.

2407 The calculated result of real short floating-point arguments for EXP overflowed the output field.

2408 The calculated result of real long floating-point arguments for EXP overflowed the output field.

2409 The calculated result of real extended floating-point arguments for EXP overflowed the output field.

2410 The calculated result of real short floating-point arguments for SCALE overflowed the output field.

2411 The calculated result of real long floating-point arguments for SCALE overflowed the output field.

2412 The calculated result of real extended floating-point arguments for SCALE overflowed the output field.

2413 Computational error; complex short floating-point argument in LOG, LOG2, or LOG10 built-in function was zero.

2414 Computational error; complex long floating-point argument in LOG, LOG2, or LOG10 built-in function was zero.

2415 Computational error; complex extended floating-point argument in LOG, LOG2, or LOG10 built-in function was zero.

2416 The calculated result of real short floating-point arguments for SINH or COSH calculated result overflowed output field

2417 The calculated result of real long floating-point arguments for SINH or COSH calculated result overflowed output field

2418 The calculated result of real extended floating-point arguments for SINH or COSH calculated result overflowed output field

2419 The calculated result of real short floating-point arguments for COTAN or COTAND calculated result overflowed output field

2420 The calculated result of real long floating-point arguments for COTAN or COTAND calculated result overflowed output field

2421 The calculated result of real extended floating-point arguments for COTAN or COTAND calculated result overflowed output field

2422 Computational error in SIN, COS, SIND, or COSD built-in function; for complex short floating-point argument the calculated result overflowed output field.

2423 Computational error in SIN, COS, SIND, or COSD built-in function; for complex long floating-point argument the calculated result overflowed output field.

2424 Computational error in SIN, COS, SIND, or COSD built-in function; for complex extended floating-point argument the calculated result overflowed output field.

2425 Computational error in SIN, COS, SIND, or COSD built-in function; real short floating-point argument is equal to plus or minus infinity.

2426 Computational error in SIN, COS, SIND, or COSD built-in function; real long floating-point argument is equal to plus or minus infinity.

2427 Computational error in TAN or TAND built-in function; real short floating-point argument equal to plus or minus infinity.

2428 Computational error in TAN or TAND built-in function; real long floating-point argument equal to plus or minus infinity.

2429 Computational error in COTAN or COTAND built-in function; real short floating-point argument is equal to plus or minus zero, or plus or minus infinity.

2430 Computational error in COTAN or COTAND built-in function; real long floating-point argument is equal to plus or minus zero, or plus or minus infinity.

2431 Computational error in COTAN or COTAND built-in function; real extended floating-point argument is equal to plus or minus zero.

2450 Computational error in EXPONENT built-in function; for complex long floating-point base with integer exponent, the calculated result was infinity.

2451 Computational error in EXPONENT built-in function; for complex extended floating-point base with integer exponent, the calculated result was infinity.

2452 Computational error in EXP built-in function; for complex short floating-point argument, the calculated result was infinity.

2453 Computational error in EXP built-in function; for complex long floating-point argument, the calculated result was infinity.

2454 Computational error in EXP built-in function; for complex extended floating-point argument, the calculated result was infinity.

2455 Computational error during division; for complex short floating-point argument, the calculated result was infinity.

2456 Computational error during division; for complex long floating-point argument, the calculated result was infinity.

2457 Computational error during division; for complex extended floating-point argument, the calculated result was infinity.

2458 Computational error in SQRT built-in function; for real short floating-point arguments, the ONCODE value was infinity.

2459 Computational error in SQRT built-in function; for real long floating-point arguments, the ONCODE value was infinity.

2460 Computational error in SQRT built-in function; for real extended floating-point arguments, the ONCODE value was infinity.

2461 Computational error in LOG built-in function; for real short floating-point arguments, the calculated result was infinity.

2462 Computational error in LOG built-in function; for real long floating-point arguments, the calculated result was infinity.

2463 Computational error in LOG built-in function; for real extended floating-point arguments, the calculated result was infinity.

2464 Computational error in ERFC built-in function; for real short floating-point arguments, the ONCODE value was infinity.

2465 Computational error in EFRC built-in function; for real long floating-point arguments, the ONCODE value was infinity.

2466 Computational error in EFRC built-in function; for real extended floating-point arguments, the ONCODE value was infinity.

2467 Computational error in ABS built-in function; for real short floating-point arguments, the ONCODE value was infinity.

2468 Computational error in ABS built-in function; for real long floating-point arguments, the ONCODE value was infinity.

2469 Computational error in ABS built-in function; for real extended floating-point arguments, the ONCODE value was infinity.

2470 Computational error in GAMMA or LOGGAMMA built-in function; for real short floating-point argument, the calculated result was infinity.

2471 Computational error in GAMMA or LOGGAMMA built-in function; for real long floating-point argument, the calculated result was infinity.

2472 Computational error in GAMMA or LOGGAMMA built-in function; for real extended floating-point argument, the calculated result was infinity.

2473 Computational error in EXPONENT built-in function; for real short floating-point base with real short floating-point exponent, the calculated result was infinity.

2474 Computational error in EXPONENT built-in function; for real long floating-point base with real long floating-point exponent, the calculated result was infinity.

2475 Computational error in EXPONENT built-in function; for real extended floating-point base with real extended floating-point exponent, the calculated result was infinity.

2476 Computational error in EXPONENT built-in function; for real short floating-point base with integer exponent, the calculated result was infinity.

2477 Computational error in EXPONENT built-in function; for real long floating-point base with integer exponent, the calculated result was infinity.

2478 Computational error in EXPONENT built-in function; for real extended floating-point base with integer exponent, the calculated result was infinity.

2479 Computational error in EXP built-in function; for real short floating-point argument, the calculated result was infinity.

2480 Computational error in EXP built-in function; for real long floating-point argument, the calculated result was infinity.

2481 Computational error in EXP built-in function; for real extended floating-point argument, the calculated result was infinity.

2513 Computational error in SQRT built-in function; for complex short floating-point arguments, the ONCODE value was infinity.

2514 Computational error in SQRT built-in function; for complex long floating-point arguments, the ONCODE value was infinity.

2515 Computational error in SQRT built-in function; for complex extended floating-point arguments, the ONCODE value was infinity.

2516 Computational error during multiplication; for complex short floating-point argument, the calculated result was infinity.

2517 Computational error during multiplication; for complex long floating-point argument, the calculated result was infinity.

2518 Computational error during multiplication; for complex extended floating-point argument, the calculated result was infinity.

2519 Computational error in LOG built-in function; for complex short floating-point arguments, the calculated result was infinity.

2520 Computational error in LOG built-in function; for complex long floating-point arguments, the calculated result was infinity.

2521 Computational error in LOG built-in function; for complex extended floating-point arguments, the calculated result was infinity.

2522 Computational error in EFRC built-in function; for complex short floating-point arguments, the ONCODE value was infinity.

2523 Computational error in EFRC built-in function; for complex long floating-point arguments, the ONCODE value was infinity.

2524 Computational error in EFRC built-in function; for complex extended floating-point arguments, the ONCODE value was infinity.

2525 Computational error in ABS built-in function; for complex short floating-point arguments, the ONCODE value was infinity.

2526 Computational error in ABS built-in function; for complex long floating-point arguments, the ONCODE value was infinity.

2527 Computational error in ABS built-in function; for complex extended floating-point arguments, the ONCODE value was infinity.

2528 Computational error in EXPONENT built-in function; for complex short floating-point base with complex short floating-point exponent, the calculated result was infinity.

2529 Computational error in EXPONENT built-in function; for complex long floating-point base with complex long floating-point exponent, the calculated result was infinity.

2530 Computational error in EXPONENT built-in function; for complex extended floating-point base with complex extended floating-point exponent, the calculated result was infinity.

2531 Computational error in EXPONENT built-in function; for complex short floating-point base with integer exponent, the calculated result was infinity.

3000 Field width, number of fractional digits, and number of significant digits (w,d, and s) specified for E-format item in edit-directed input/output statement do not allow transmission without loss of significant digits or sign.

3006 Picture description of target does not match noncharacter-string source.

3009 A mixed character string contained a shift-out, then ended before a shift-in was found.

3010 During processing of a mixed character constant, one of the following occurred:

- A shift-in present in the SBCS portion.
- A shift-out present in the graphic (double-byte) portion. (A shift-out cannot appear in either byte of a graphic character).
- A shift-in present in the second byte of a graphic character.

3011 MPSTR built-in function contains an invalid character (or a null function string, or only blanks) in the expression that specifies processing rules. (Only V, v, S, s, and blank are valid characters).

3012 Retry for graphic conversion error not allowed.

3013 An assignment attempted to a graphic target with a length greater than 16,383 characters (32,766 bytes).

3014 A graphic or mixed string did not conform to the continuation rules.

3015 A *X* or *GX* constant has an invalid number of digits.

3016 Improper use of graphic data in stream I/O. Graphic data can only be used as part of a variable name or string.

3017 Invalid graphic, mixed, or DBCS continuation when writing Stream I/O to a file containing fixed-length records.

3797 Attempt to convert to or from graphic data.

3798 ONCHAR or ONSOURCE pseudovariable used out of context.

3799 The source was not modified in the CONVERSION ON-unit. Retry was not attempted. An ON-unit was entered as a result of the CONVERSION condition being raised by an invalid character in the string being converted. The character was not corrected in an ON-unit using the ONSOURCE or ONCHAR pseudovariables.

3800 Length of data aggregate exceeds system limit of 2**24 bytes.

3808 Aggregate cannot be mapped in COBOL or FORTRAN.

3809 A data aggregate exceeded the maximum length.

3810 An array has an extent that exceeds the allowable maximum.

3901 Attempt to invoke process using a process variable that is already associated with an active process.

3904 Event variable referenced as argument to COMPLETION pseudovariable while already in use for a DISPLAY statement.

3906 Assignment to an event variable that is already active.

3907 Attempt to associate an event variable that is already associated with an active process.

3909 Attempt to create a subtask (using CALL statement) when insufficient main storage available.

3910 Attempt to attach a process (using CALL statement) when number of active processes was already at limit defined by ISASIZE parameter of EXEC statement.

3911 WAIT statement in ON-unit references an event variable already being waited for in process from which ON-unit was entered.

3912 Attempt to execute CALL with TASK option in block invoked while executing PUT FILE(SYSPRINT) statement.

3913 CALL statement with TASK option specifies an unknown entry point.

3914 Attempt to call FORTRAN or COBOL routines in two processes simultaneously.

3915 Attempt to call a process when the multitasking library was not selected in the link-edit step.

3920 An out-of-storage abend occurred.

8091 Operation exception.

8092 Privileged operation exception.

8093 EXECUTE exception.

8094 Protection exception.

8095 Addressing exception.

8096 Specification exception.

8097 Data exception.

9002 Attempt to execute GO TO statement referencing label in an inactive block.

9003 Attempt to execute a GO TO statement to a non-existent label constant

9050 Program terminated by an abend.

9200 Program check in SORT/MERGE program.

9201 SORT not supported in CMS.

9250 Procedure to be fetched cannot be found.

9251 Permanent transmission error when fetching a procedure.

9252 FETCH/RELEASE not supported in CMS.

9253 PLITEST unavailable.

9999 A failure occurred in invocation of a Language Environment service. Use the ONFEEDBACK built-in function to determine the precise Language Environment error.

Chapter 17. Built-In Functions, Pseudovariables, and Subroutines

Chapter 17. Built-in functions, pseudovariables, and subroutines

A large number of common tasks are available in the form of built-in functions, subroutines and pseudovariables. Using them lets you write less code more quickly with greater reliability.

The built-in functions, subroutines, and pseudovariables are listed in alphabetic order later in this chapter. In general, each description has the following format:

- A heading showing the syntax of the reference
- A description of the value returned or, for a pseudovariable, the value set
- A description of any arguments
- Any other qualifications on using the function or pseudovariable.

Note that the abbreviations for built-in functions have separate declarations (explicit or contextual) and name scopes. In the following example:

```
dcl (Dim, Dimension) builtin;
```

is not a multiple declaration, and

```
dcl Binary file;
X = Bin (var, 6,3);
```

is valid even though *Bin* is an abbreviation of the *Binary* built-in function.

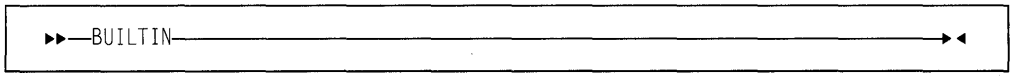
Declaring built-in functions

Built-in functions, pseudovariables, and subroutines can be contextually or explicitly declared.

BUILTIN attribute

The BUILTIN attribute specifies that the name is a built-in function, pseudovariable, or a subroutine.

The syntax of the BUILTIN attribute is:



Built-in names can be used as programmer-defined names. BUILTIN can be declared for a built-in name in any block that has inherited, from a containing block, a programmer-defined declaration or use of the same name. The following examples show built-in names with the BUILTIN attribute.

Example 1

```
|1|   A: procedure;
         declare Sqrt float binary;
|2|      X = Sqrt;

|3|     B:  Begin;
           Declare Sqrt builtin;
           Z = Sqrt(P);
         end B;

       end A;
```

Example 2

```
    A:  procedure;
|1|   Sqrt: proc(Param) returns(fixed(6,2));
          declare Param fixed (12);
          end Sqrt;

|2|       X = Sqrt(Y);

|3|        B:  begin;
             declare Sqrt builtin;
             Z = Sqrt (P);
           end B;

    end A;
```

In both examples:

|1| `Sqrt` is a programmer-defined name.

|2| The assignment to the variable `X` is a reference to the programmer-defined name `Sqrt`.

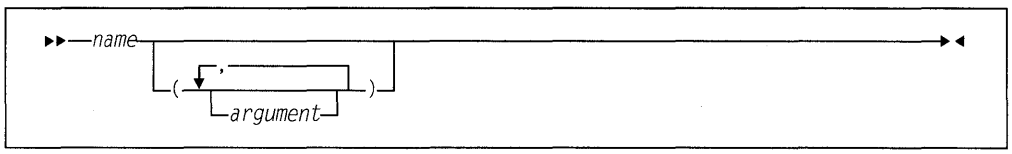
|3| `Sqrt` is declared with the BUILTIN attribute so that any reference to `Sqrt` within `B` is recognized as a reference to the built-in function and not to the programmer-defined name `Sqrt` declared in **1**.

In Example 2, if the procedure `Sqrt` is external, procedure `A` must declare `Sqrt` explicitly as an entry name, and to specify the attributes of the values passed to and returned from this programmer-written function procedure. The declaration for Example 2 would be:

```
dcl Sqrt entry (fixed (12))
        returns (fixed(6,2));
```

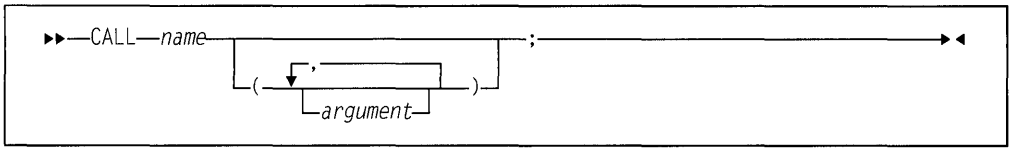
Invoking built-in functions and pseudovariables

The syntax used to invoke built-in functions and pseudovariables is as follows:



Invoking built-in subroutines

The syntax used to invoke built-in subroutines is as follows:



Specifying arguments for built-in functions

Arguments, which can be expressions, are evaluated and converted to a data type suitable for the built-in function according to the rules for data conversion.

Aggregate arguments

All built-in functions and pseudovariables that can have arguments can have array arguments (if more than one is an array, the bounds must be identical). ADDR, ALLOCATION, CURRENTSIZE, SIZE, STRING, and the array-handling functions return an element value. All other functions return an array of values. Specifying an array argument is equivalent to placing the function reference or pseudovariable in a do-group where one or more arguments is a subscripted array reference that is modified by the control variable.

For example:

```
dcl A(2) char(2) varying;
dcl B(2) char(2)
   init('AB','CD');
dcl C(2) fixed bin
   init(1,2);
A=substr(B,1,C);
```

results in `A(1)` having the value `A` and `A(2)` having the value `CD`.

The built-in functions and pseudovariables that can accept structure or union arguments are ADDR, ALLOCATION, CURRENTSIZE, SIZE, STRING, and UNSPEC.

Null and optional arguments

Some built-ins do not require arguments. You must either explicitly declare these with the BUILTIN attribute or contextually declare them by including a null argument list in the reference—for example, ONCHAR(). Otherwise, the name is not recognized as a built-in.

Accuracy of mathematical functions

The accuracy of a result is influenced by two factors:

- The accuracy of the argument
- The accuracy of the algorithm.

Most arguments contain errors. An error in a given argument can accumulate over several steps before the evaluation of a function. Even data fresh from input conversion can contain errors. The effect of argument error on the accuracy of a result depends entirely on the nature of the mathematical function, and not on the algorithm that computes the result. This book does not discuss argument errors of this type.

Categories of built-in functions

The following sections list built-in functions, subroutines, and pseudovariables.

Only full function names are listed in these tables. Abbreviation that exist are provided in the sections that describe each built-in function, subroutine, and pseudovariable.

Arithmetic built-in functions

The arithmetic built-in functions allow you to do the following:

- Determine properties of arithmetic values. For example, the SIGN function indicates the sign of an arithmetic variable.
- Perform routine arithmetic operations.

Figure 54 lists the arithmetic built-in functions.

Some of the arithmetic functions derive the data type of their results from one or more arguments. When the data types of the arguments differ, they are converted as described in Chapter 5, "Data conversion" on page 72. When a data attribute for the result cannot agree with that of the argument, the rules are given in the function description.

Figure 54. Arithmetic built-in functions

Function	Description
ABS	Calculates the absolute value of a value
CEIL	Calculates the smallest integer value greater than or equal to a value
COMPLEX	Returns the complex number with given real and imaginary parts
CONJG	Returns the complex conjugate of a value
FLOOR	Calculates the largest integer value less than or equal to a value
IMAG	Returns the imaginary part of a complex number
MAX	Calculates the maximum of 2 or more values
MIN	Calculates the minimum of 2 or more values
MOD	Returns the modular equivalent of the remainder of one value divided by another
RANDOM	Returns a pseudo-random value
REM	Calculates the remainder of one value divided by another
REAL	Returns the real part of a complex number
ROUND	Rounds a value at a specified digit
SIGN	Returns a -1, 0 or 1 if a value is negative, zero or positive respectively
TRUNC	Calculates the nearest integer for value rounded towards zero

Array-handling built-in functions

The array-handling built-in functions operate on array arguments and return an element value. Any conversion of arguments required for these functions is noted in the function description. Figure 55 lists the array-handling built-in functions.

Figure 55. Array-handling built-in functions

Function	Description
ALL	Calculates the bitwise "and" of all the elements of an array
ANY	Calculates the bitwise "or" of all the elements of an array
DIMENSION	Returns the number of elements in a dimension of an array
HBOUND	Returns the upper bound for a dimension of an array
LBOUND	Returns the lower bound for a dimension of an array
PROD	Calculates the product of all the elements of an array
SUM	Calculates the sum of all the elements of an array

Condition-handling built-in functions

The condition-handling built-in functions allow you to determine the cause of a condition that has occurred.

Use of these functions is valid only within the scope of an ON-unit or dynamic descendant for the condition specific to the built-in function, or within an ON-unit or a dynamic descendant for the ERROR or FINISH condition when raised as an implicit action. All other uses are out of context.

Figure 56. Condition-handling built-in functions

Function	Description
DATAFIELD	Returns the value of a string that raised the NAME condition
ONCHAR	Returns the value of a character that caused a conversion condition
ONCODE	Returns the condition code value
ONCOUNT	Returns the number of outstanding conditions
ONFILE	Returns the name of a file for which a condition is raised
ONGSOURCE	Returns a double-byte DBCS character string containing the DBCS character that caused the CONVERSION condition to be raised.
ONKEY	Returns the key of a record that raised a condition
ONLOC	Returns the name of the procedure in which a condition occurred
ONSOURCE	Returns the value of a string that caused a conversion condition

Date/time built-in functions

The date/time built-in functions return date and time information as a timestamp. Figure 57 lists the supported date/time built-in functions.

Figure 57 (Page 1 of 2). Date/time built-in functions

Function	Description
DATE	Returns the current date in the pattern YYMMDD

Figure 57 (Page 2 of 2). Date/time built-in functions

Function	Description
DATETIME	Returns the current date and time in the pattern YYYYMMDDHHMISS999
TIME	Returns the current time in the pattern HHMMSS999

Floating-point inquiry built-in functions

The floating-point inquiry built-in functions return information about a the floating-point variable arguments that you specify. Figure 58 lists these built-in functions.

Figure 58. Floating-point inquire built-in functions

Function	Description
EPSILON	Returns the spacing around 1
MAXEXP	Returns the maximum value for an exponent
MINEXP	Returns the minimum value for an exponent
HUGE	Returns the largest positive finite value that a floating-point variable can hold
PLACES	Returns the model precision for a floating point value
RADIX	Returns the model base for a floating point value
TINY	Returns the smallest positive value that a floating-point variable can hold

Floating-point manipulation built-in functions

The floating-point manipulation built-in functions perform mathematical operations on floating-point variables that you specify and return the result of the operation. Figure 59 lists the floating-point manipulation functions.

Figure 59. Floating-point manipulation built-in functions

Function	Description
EXPONENT	Returns the exponent part of a floating point value
SCALE	Multiplies a floating-point number by an integral power of 2
PRED	Returns the next representable value before a floating-point value
SUCC	Returns the next representable value after a floating-point value

Input/output built-in functions

The input/output built-in functions allow you to determine the current state of a file. Figure 60 lists these functions.

Figure 60 (Page 1 of 2). Input/output built-in functions

Function	Description
COUNT	Returns the number of data items transmitted during the last GET or PUT
ENDFILE	Indicates if a file is open and end-of-file has been reached for it

Figure 60 (Page 2 of 2). Input/output built-in functions

Function	Description
FILEOPEN	Indicates if a file is open
LINENO	Returns the current line number associated with a print file
PAGENO	Returns the current page number associated with a print file
SAMEKEY	Indicates if a record is followed by another with the same key

Integer manipulation built-in functions

The integer manipulation built-in functions perform operations on integer variables and return the result of the operation. Figure 61 lists the integer manipulation functions.

Figure 61. Integer manipulation built-in functions

Function	Description
IAND	Calculates the bitwise "and" of 2 fixed binary values
IEOR	Calculates the bitwise "exclusive-or" of 2 fixed binary values
IOR	Calculates the bitwise "or" of 2 fixed binary values
LOWER2	Divides a fixed binary value by an integral power of 2
RAISE2	Multiplies a fixed binary value by an integral power of 2

Mathematical built-in functions

All of these functions operate on floating-point values to produce a floating-point result. Any argument that is not floating-point is converted. The accuracy of these functions is discussed in "Accuracy of mathematical functions" on page 373. Figure 62 lists the mathematical built-in functions.

Figure 62 (Page 1 of 2). Mathematical built-in functions

Function	Description
ACOS	Calculates the arc cosine
ASIN	Calculates the arc sine
ATAN	Calculates the arc tangent
ATAND	Calculates the arc tangent in degrees
ATANH	Calculates the hyperbolic arc tangent
COS	Calculates the cosine
COSD	Calculates the cosine for a value in degrees
COSH	Calculates the hyperbolic cosine
COTAN	Calculates the cotangent
COTAND	Calculates the cotangent for a value in degrees
ERF	Calculates the error function
ERFC	Calculates the complement of the error function
EXP	Calculates e to a power
GAMMA	Calculates the gamma function

Figure 62 (Page 2 of 2). Mathematical built-in functions

Function	Description
LOG	Calculates the natural logarithm
LOGGAMMA	Calculates the log of the gamma function
LOG10	Calculates the base 10 logarithm
LOG2	Calculates the base 2 logarithm
SIN	Calculates the sine
SIND	Calculates the sine for a value in degrees
SINH	Calculates the hyperbolic sine
SQRT	Calculates the square root
TAN	Calculates the tangent
TAND	Calculates the tangent for a value in degrees
TANH	Calculates the hyperbolic tangent

Miscellaneous built-in functions

The built-in functions that do not fit into any of the previous categories are those listed in Figure 63.

Figure 63. Miscellaneous built-in functions

Function	Description
COLLATE	Returns a character(256) string specifying the collating order
COMPARE	Compares n bytes at two addresses
HEX	Returns a character string that is the hex representation of a value
HEXIMAGE	Returns a character string that is the hex representation of a specified number of bytes at a given address
OMITTED	Indicates if a parameter was not supplied on a call
PLIRETV	Returns the PL/I return code value
STRING	Returns a string that is the concatenation of all the elements of a string aggregate
UNSPEC	Returns a bit string that is the internal representation of a value
VALID	Indicates if the contents of a variable are valid for its declaration

Precision-handling built-in functions

The precision-handling built-in functions allow you to manipulate variables with specified precisions, and they return the value resulting from the operation.

Figure 64 (Page 1 of 2). Precision-handling built-in functions

Function	Description
ADD[1]	Adds, with a specified precision, two values
BINARY	Converts a value to binary
DECIMAL	Converts a value to decimal
DIVIDE	Divides, with a specified precision, two values

Figure 64 (Page 2 of 2). Precision-handling built-in functions

Function	Description
FIXED	Converts a value to fixed
FLOAT	Converts a value to float
MULTIPLY	Multiplies, with a specified precision, two values
SIGNED	Converts a value to signed fixed binary
PRECISION	Converts a value to specified precision
SUBTRACT	Subtracts, with a specified precision, two values
UNSIGNED	Converts a value to unsigned fixed binary

Pseudovariables

Pseudovariables represent receiving fields. They cannot be nested, and can be used only on the left side of an assignment statement. For example, the following is invalid:

```
unspec(substr(A,1,2)) = '00'B;
```

The pseudovariables are:

Figure 65. Built-in pseudovariables

Function	Description
ENTRYADDR	Sets an entry variable with the address of the entry to be invoked.
IMAG	Assigns the imaginary part of a complex number
ONCHAR	Sets the value of a character that caused a conversion condition
ONGSOURCE	Sets a double-byte DBCS character string containing the DBCS character that caused the CONVERSION condition to be raised.
ONSOURCE	Sets the value of a string that caused a conversion condition
REAL	Assigns the real part of a complex number
STRING	Assigns a string that is the concatenation of all the elements of a string aggregate
SUBSTR	Assigns a substring of a string
UNSPEC	Assigns a bit string that is the internal representation of a value

Storage control built-in functions

The storage control built-in functions allow you to determine the storage requirements and location of variables, to assign special values to area and locator variables, to perform conversion between offset and pointer values, and to obtain the number of generations of a controlled variable. Figure 66 lists the storage control built-in functions.

Figure 66 (Page 1 of 2). Storage control built-in functions

Function	Description
ADDR	Returns the address of a variable
ALLOCATION	Returns the current number of generations of a controlled variable
BINARYVALUE	Converts a pointer or offset to an integer

Figure 66 (Page 2 of 2). Storage control built-in functions

Function	Description
CURRENTSIZE	Returns the current size of a variable
EMPTY	Returns an "empty" area
ENTRYADDR	Returns the address of the routine associated with an entry
NULL	Returns a null pointer value
OFFSET	Converts a pointer to an offset
OFFSETADD	Adds an integer to an offset
OFFSETDIFF	Subtracts two offsets
OFFSETSUBTRACT	Subtracts an integer from an offset
OFFSETVALUE	Converts an integer to an offset
POINTER	Converts an offset to a pointer
POINTERADD	Adds an integer to a pointer
POINTERDIFF	Subtracts two pointers
POINTERSUBTRACT	Subtracts an integer from a pointer
POINTERVALUE	Converts an integer to a pointer
SIZE	Returns the maximum size of a variable
SYSNULL	Returns a system null pointer value

String-handling built-in functions

The string-handling built-in functions simplify the processing of bit, character, and DBCS strings. The character- and bit-string arguments can be represented by an arithmetic expression that will be converted to string either according to data conversion rules or according to the rules given in the function description.

Note: The functions CENTERLEFT, CENTERRIGHT, LEFT, RIGHT, TRANSLATE, and TRIM do not support GRAPHIC data.

Figure 67 (Page 1 of 2). String-handling built-in functions

Function	Description
BIT	Converts a value to bit
BOOL	Performs boolean operation on 2 bit strings
CENTERLEFT	Returns a string with a value centered (to the left) in it
CENTERRIGHT	Returns a string with a value centered (to the right) in it
CENTRELEFT	Returns a string with a value centered (to the left) in it
CENTRERIGHT	Returns a string with a value centered (to the right) in it
CHARACTER	Converts a value to character
COPY	Returns a string consisting of n copies of a string
GRAPHIC	Converts a value to graphic
HIGH	Returns a string consisting of n copies of the highest character in the collating sequence
INDEX	Finds the location of one string within another
LEFT	Returns a string with a value left-justified in it

Figure 67 (Page 2 of 2). String-handling built-in functions

Function	Description
LENGTH	Returns the current length of a string
LOW	Returns a string consisting of n copies of the lowest character in the collating sequence
MAXLENGTH	Returns the maximum length of a string
MPSTR	Truncates a string at a logical boundary and returns a mixed character string.
REPEAT	Returns a string consisting of n+1 copies of a string
REVERSE	Returns a reversed image of a string
RIGHT	Returns a string with a value right-justified in it
SEARCH	Searches for the first occurrence of any one of the elements of a string within another string
SEARCHR	Searches for the first occurrence of any one of the elements of a string within another string but the search starts from the right
SUBSTR	Assigns a substring of a string
TRANSLATE	Translates a string based on two translation strings
TRIM	Trims specified characters from the left and right sides of a string
VERIFY	Searches for first non-occurrence of any one of the elements of a string within another string
VERIFYR	Searches for first non-occurrence of any one of the elements of a string within another string but the search starts from the right

Subroutines

Built-in subroutines perform miscellaneous operations that do not necessarily return a result as built-in functions do. Figure 68 lists the built-in subroutines.

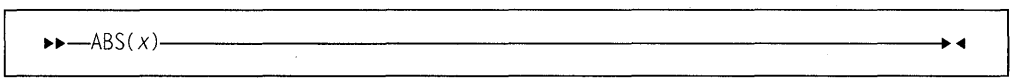
Figure 68. Built-in subroutines

Function	Description
PLIDUMP	Dumps information about currently open files, the calling path to the current location, etc
PLIFILL	Fills n bytes at an address with a specified byte value
PLIMOVE	Moves n bytes from one address to another
PLIRETC	Sets the PL/I return code value

ABS

ABS returns the absolute value of *x*. It is the positive value of *x*.

The syntax for ABS is:



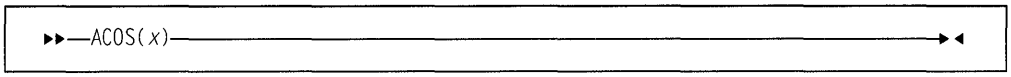
x expression.

The mode of the result is REAL. The result has the base, scale, and precision of *x*. The precision for FIXED COMPLEX is min(N,P+1).

ACOS

ACOS returns a real floating-point value that is an approximation of the inverse (arc) cosine in radians of *x*.

The syntax for ACOS is:



x real expression, where ABS(x) <= 1.

The result is in the range:

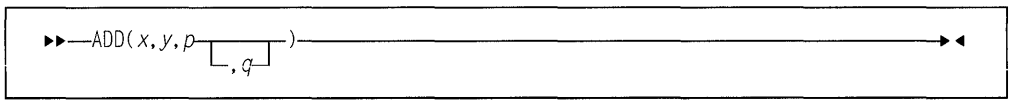
$0 \leq \text{ACOS}(x) \leq \pi$

and has the base and precision of *x*.

ADD

ADD returns the sum of *x* and *y* with a precision specified by *p* and *q*. The base, scale, and mode of the result are determined by the rules for expression evaluation.

The syntax for ADD is:



x and y
expressions.

p restricted expression. It specifies the number of digits to be maintained throughout the operation.

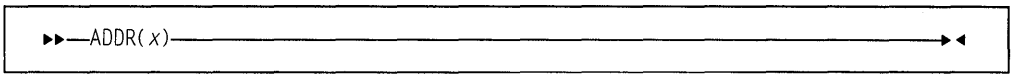
q restricted expression specifying the scaling factor of the result. For a fixed-point result, if *q* is omitted, a scaling factor of zero is the default. For a floating-point result, *q* must be omitted.

ADD can be used for subtraction by prefixing a minus sign to the operand to be subtracted.

ADDR

ADDR returns the pointer value that identifies the generation of *x*.

The syntax for ADDR is:



x reference. It refers to a variable of any data type, data organization, alignment, and storage class except:

- A subscripted reference to a variable that is an unaligned fixed-length bit string
- A reference to a DEFINED or BASED variable or simple parameter, which is an unaligned fixed-length bit string
- A minor structure or union whose first base element is an unaligned fixed-length bit string (except where it is also the first element of the containing major structure or union)
- A major structure or union that has the DEFINED attribute or is a parameter, and that has an unaligned fixed-length bit string as its first element
- A reference that is not to connected storage.

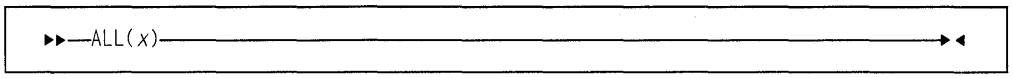
If *x* is a reference to:

- An aggregate parameter, it must have the CONNECTED attribute.
- An aggregate, the returned value identifies the first element
- A component or cross section of an aggregate, the returned value takes into account subscripting and structure or union qualification
- A varying string, the returned value identifies the 2-byte prefix
- An area, the returned value identifies the control information
- A controlled variable that is not allocated in the current program, the null pointer value is returned
- A based variable, the result is the value of the pointer explicitly qualifying *x* (if it appears), or associated with *x* in its declaration (if it exists), or a null pointer
- A parameter, and a dummy argument has been created, the returned value identifies the dummy argument.

ALL

ALL returns a bit string in which each bit is 1 if the corresponding bit in each element of *x* exists and is 1. The length of the result is equal to that of the longest element.

The syntax for ALL is:

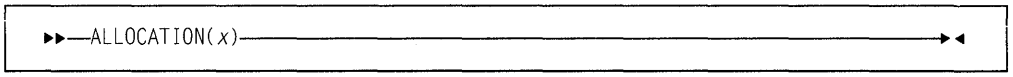


x array expression that must be either a NONVARYING BIT array reference or an expression that compares an array reference and an expression.

ALLOCATION

ALLOCATION returns a FIXED BIN(M,0) value specifying the number of generations that can be accessed in the current program for *x*.

The syntax for ALLOCATION is:



Abbreviation: ALLOCN

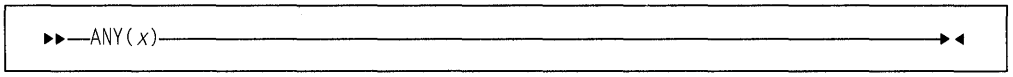
x level-one unsubscripted controlled variable.

If *x* is not allocated in the current program, the result is zero.

ANY

ANY returns a bit string in which each bit is 1 if the corresponding bit in any element of *x* exists and is 1. The length of the result is equal to that of the longest element.

The syntax for ANY is:

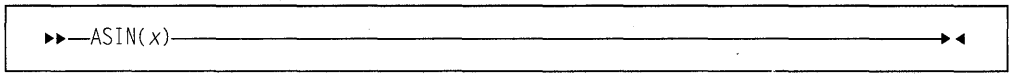


x array expression that must be either a NONVARYING BIT array reference or an expression that compares an array reference and an expression.

ASIN

ASIN returns a real floating-point value that is an approximation of the inverse (arc) sine in radians of *x*.

The syntax for ASIN is:



x real expression, where ABS(x) <= 1.

The result is in the range:

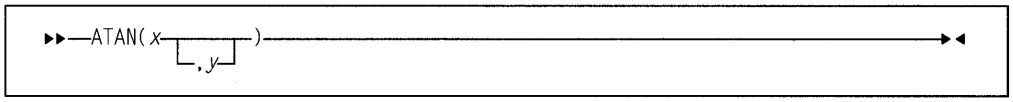
$-\pi/2 \leq \text{ASIN}(x) \leq \pi/2$

and has the base and precision of *x*.

ATAN

ATAN returns a floating-point value that is an approximation of the inverse (arc) tangent in radians of *x* or of a ratio *x/y*.

The syntax for ATAN is:



x and y

expressions.

If *x* alone is specified, the result has the base and precision of *x*, and is in the range:

$-\pi/2 < \text{ATAN}(x) < \pi/2$

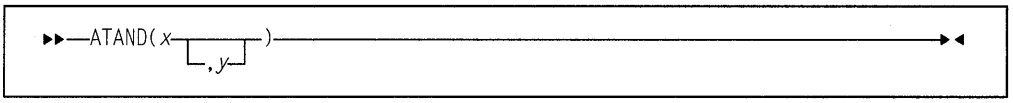
If *x* and *y* are specified, each must be real. An error exists if *x* and *y* are both zero. The result for all other values of *x* and *y* has the precision of the longer argument, a base determined by the rules for expressions, and a value given by:

ATAN(*x*/*y*)	for y>0
$\pi/2$	for y=0 and x>0
$-\pi/2$	for y=0 and x<0
π + ATAN(*x*/*y*)	for y<0 and x>=0
$-\pi$ + ATAN(*x*/*y*)	for y<0 and x<0

ATAND

ATAND returns a real floating-point value that is an approximation of the inverse (arc) tangent in degrees of *x* or of a ratio *x/y*.

The syntax for ATAND is:



x and y

expressions.

If *x* alone is specified it must be real. The result has the base and precision of *x*, and is in the range:

$-90 < \text{ATAND}(x) < 90$

If *x* and *y* are specified, each must be real. The value of the result is given by:

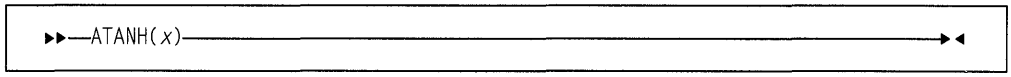
$(180/\pi)*\text{ATAN}(x,y)$

For argument requirements and attributes of the result see "ATAN" on page 385.

ATANH

ATANH returns a floating-point value that has the base, mode, and precision of *x*, and is an approximation of the inverse (arc) hyperbolic tangent of *x*.

The syntax for ATANH is:



x expression. ABS(x)<1.

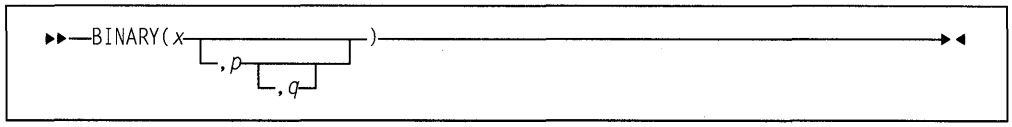
The result has a value given by:

$\text{LOG}((1 + x)/(1 - x))/2$

BINARY

BINARY returns the binary value of *x*, with a precision specified by *p* and *q*. The result has the mode and scale of *x*.

The syntax for BINARY is:



Abbreviation: BIN

x expression.

p restricted expression. Specifies the number of digits to be maintained throughout the operation; it must not exceed the implementation limit.

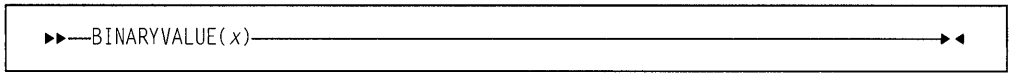
q restricted expression. It specifies the scaling factor of the result. For a fixed-point result, if p is given and *q* is omitted, a scaling factor of zero is the default. For a floating-point result, *q* must be omitted.

If both *p* and *q* are omitted, the precision of the result is determined from the rules for base conversion.

BINARYVALUE

BINARYVALUE returns a FIXED BIN(M,0) value that is the converted value of its pointer expression, *x*.

The syntax for BINARYVALUE is:



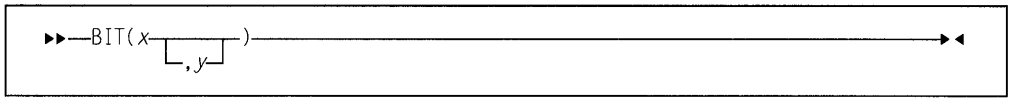
Abbreviation: BINVALUE

x expression.

BIT

BIT returns the bit value of *x*, with a length specified by *y*.

The syntax for BIT is:



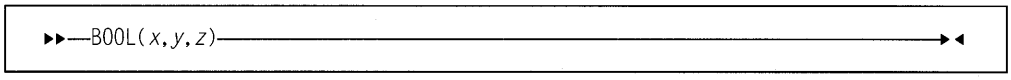
x expression.

y expression. If necessary, *y* is converted to a real fixed-point binary value. If *y* is omitted, the length is determined by the rules for type conversion. If *y* = 0, the result is the null bit string. *y* must not be negative.

BOOL

BOOL returns a bit string that is the result of the Boolean operation *z*, on *x* and *y*. The length of the result is equal to that of the longer operand, *x* or *y*.

The syntax for BOOL is:



x and y

expressions. *x* and *y* are converted to bit strings, if necessary. If *x* and *y* are of different lengths, the shorter is padded on the right with zeros to match the longer.

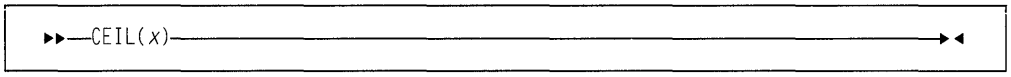
z expression. *z* is converted to a bit string of length 4, if necessary. When a bit from *x* is matched with a bit from *y*, the corresponding bit of the result is specified by a selected bit of *z*, as follows:

x	y	Result
0	0	bit 1 of z
0	1	bit 2 of z
1	0	bit 3 of z
1	1	bit 4 of z

CEIL

CEIL determines the smallest integer value greater than or equal to *x*, and assigns this value to the result.

The syntax for CEIL is:



x real expression.

The result has the mode, base, scale, and precision of *x*, except when *x* is fixed-point with precision (*p,q*). The precision of the result is then given by:

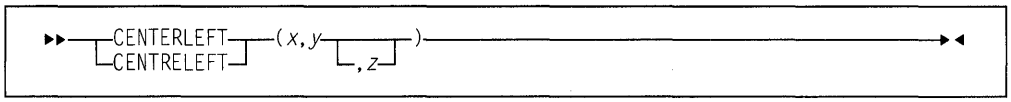
```
(min(N,max(p-q+1,1)),0)
```

where N is the maximum number of digits allowed.

CENTERLEFT

CENTERLEFT returns a string that is the result of inserting string *x* in the center (or one position to the left of center) of a string with length y and padded on the left and on the right with the character *z* as needed. If *z* is omitted, a blank is used as the padding character.

The syntax for CENTERLEFT is:



Abbreviation: CENTER

x expression that is converted to character.

y expression that is converted to FIXED BIN(M,0).

z optional expression. If specified, *z* must be CHARACTER(1) NON-VARYING type.

Example

```
dcl source char value('Feel the Power');
dcl target20 char(20);
dcl target21 char(21);

target20 = center (source, length(target20), '*');
           /* '***Feel the Power***' - exactly centered */

target21 = center (source, length(target21), '*');
           /* '***Feel the Power****' - leaning left! */
```

CENTRELEFT

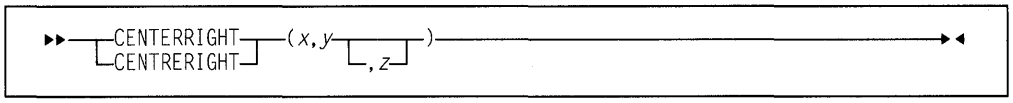
Abbreviation: CENTRE

CENTRELEFT is a synonym for CENTERLEFT.

CENTERRIGHT

CENTERRIGHT returns a string that is the result of inserting string *x* in the center (or one position to the right of center) of a string with length *y* and padded on the left and on the right with the character *z* as needed. If *z* is omitted, a blank is used as the padding character.

The syntax for CENTERRIGHT is:



x expression that is converted to character.

y expression that is converted to FIXED BIN(M,0).

z optional expression. If specified, *z* must be CHARACTER(1) NON-VARYING type.

Example

```
dcl source char value('Feel the Power');
dcl target20 char(20);
dcl target21 char(21);

target20 = centerright (source, length(target20), '*');
           /* '***Feel the Power***' - exactly centered */

target21 = centerright (source, length(target21), '*');
           /* '****Feel the Power***' - leaning right! */
```

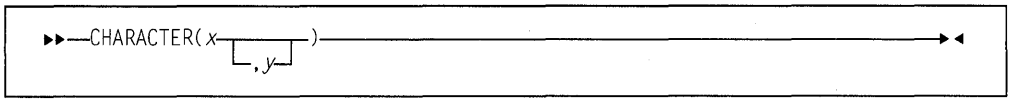
CENTRERIGHT

CENTRERIGHT is a synonym for CENTERRIGHT.

CHARACTER

CHARACTER returns the character value of *x*, with a length specified by *y*. CHARACTER also supports conversion from graphic to character type.

The syntax for CHARACTER is:



Abbreviation: CHAR

x expression.

x must have a computational type.

When *x* is nongraphic, CHARACTER returns *x* converted to character.

When *x* is GRAPHIC, CHARACTER returns *x* converted to mixed character.

The values of *x* are not checked.

y expression. If necessary, y is converted to a real fixed-point binary value.

If *y* is omitted, the length is determined by the rules for type conversion.

y cannot be negative.

If *y* = 0, the result is the null character string.

The following apply only when *x* is GRAPHIC:

If *y* = 1, the result is a character string of 1 blank.

If *y* is greater than the length needed to contain the character string, the result is padded with SBCS blanks.

If *y* is less than the length needed to contain the character string, the result is truncated. The integrity is preserved by truncating after a graphic, and appending an SBCS blank if necessary, to complete the length of the string.

Example 1: Conversion from graphic to character, where "y" is long enough to contain the result:

```
dcl X graphic(6);
dcl A char (14);
A = char(X);
```

For X with value:	Intermediate Result:	A is assigned:
.A.B.C.D.E.F	.A.B.C.D.E.F	.A.B.C.D.E.F

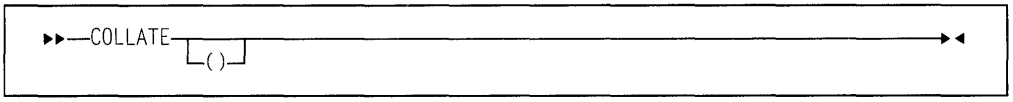
Example 2: Conversion from graphic to character, where "y" is too short:

For X with value:	Intermediate Result:	A is assigned:
.A.B.C.D.E.F	.A.B.C.D.E.F	.A.B.C.D.E♁

COLLATE

COLLATE returns a CHARACTER(256) string comprising the 256 possible CHARACTER(1) values one time each in the collating order.

The syntax for COLLATE is:

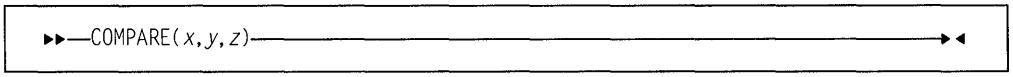


COMPARE

COMPARE returns a FIXED BIN(M,0) string that is:

- Zero, if the z bytes at the addresses x and y are identical
- Negative, if the z bytes at x are less than those at y
- Positive, if the z bytes at x are greater than those at y.

The syntax for COMPARE is:



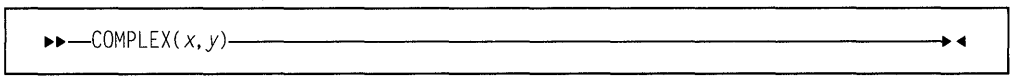
x and y
expressions. Both must have the POINTER or OFFSET type. If OFFSET, the expression must be declared with the AREA qualification.

z expression that is converted to FIXED BIN(M,0).

COMPLEX

COMPLEX returns the complex value $x + yI$.

The syntax for COMPLEX is:



Abbreviation: CPLX

x and y
real expressions.

If *x* and *y* differ in base, the decimal argument is converted to binary. If they differ in scale, the fixed-point argument is converted to floating-point. The result has the common base and scale.

The precision of the result, if fixed-point, is given by:

```
(min(N,max(p1-q1,p2-q2)+max(q1,q2)),max(q1,q2))
```

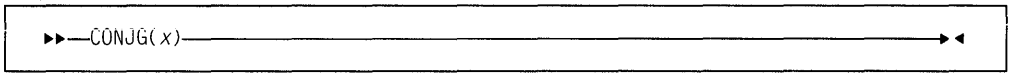
where (p1,q1) and (p2,q2) are the precisions of *x* and *y*, respectively, and N is the maximum number of digits allowed.

After any necessary conversions have been performed, if the arguments are floating-point, the result has the precision of the longer argument.

CONJG

CONJG returns the conjugate of *x*: the value of the expression with the sign of the imaginary part reversed.

The syntax for CONJG is:



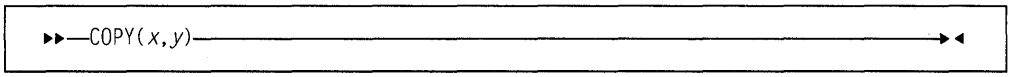
x expression.

If *x* is real, it is converted to complex. The result has the base, scale, mode, and precision of *x*.

COPY

COPY returns a string consisting of *y* concatenated copies of the string *x*.

The syntax for COPY is:



x expression.

x must have a computational type and should have a string type. If not, it is converted to character.

y an integer expression with a nonnegative value. It specifies the number of repetitions. It must have a computational type and is converted to FIXED BIN(M,0).

If *y* is zero, the result is a null string.

For example:

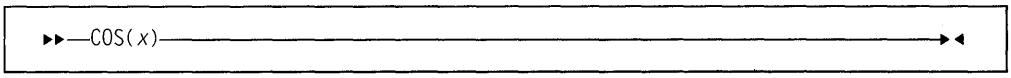
```
copy('Walla  ',1)        /*  returns 'Walla  '  */

repeat('Walla  ',1)      /*  returns 'Walla  Walla  '  */
```

`repeat(x,n)` is equivalent to `copy(x,n+1)`.

COS

COS returns a floating-point value that has the base, precision, and mode of *x*, and is an approximation of the cosine of *x*.

The syntax for COS is:

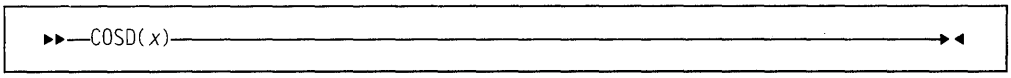


x expression with a value in radians.

COSD

COSD returns a real floating-point value that has the base and precision of *x*, and is an approximation of the cosine of *x*.

The syntax for COSD is:

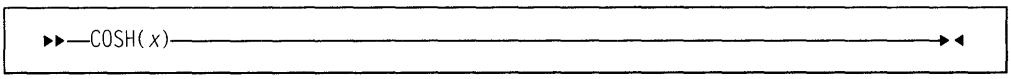


x real expression with a value in degrees.

COSH

COSH returns a floating-point value that has the base, precision, and mode of *x*, and is an approximation of the hyperbolic cosine of *x*.

The syntax for COSH is:

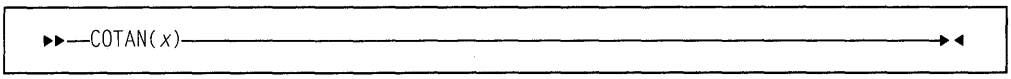


x expression.

COTAN

COTAN returns a floating-point value that has the base, mode, and precision of *x*, and is an approximation of the cotangent of *x*.

The syntax for COTAN is:



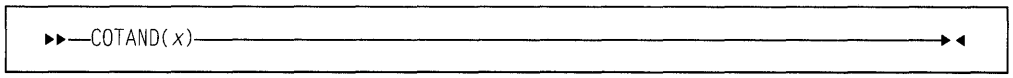
x expression. *x* must have a computational, arithmetic type. If *x* is numeric, it must be real. If *x* is not float, it is converted to float.

The value of *x* is in radians.

COTAND

COTAND returns a floating-point value that is an approximation of the cotangent of *x*. It has the base, mode, and precision of *x*.

The syntax for COTAND is:



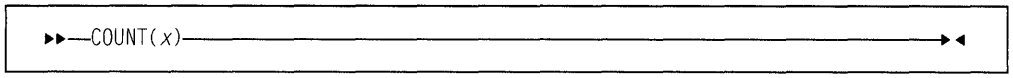
x expression. *x* must have a computational, arithmetic type. If *x* is numeric, it must be real. If *x* is not float, it is converted to float.

The value of *x* is in degrees.

COUNT

COUNT returns a real binary fixed-point value specifying the number of data items transmitted during the last GET or PUT operation on *x*.

The syntax for COUNT is:



x file-reference. The file must be open and have the STREAM attribute.

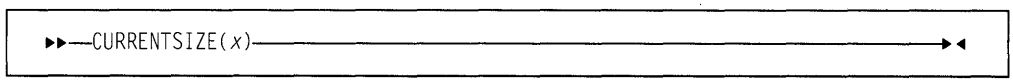
The count of transmitted items for a GET or PUT operation on *x* is initialized to zero before the first data item is transmitted, and is incremented by one after the transmission of each data item in the list. If *x* is not open in the current program, a value of zero is returned.

If an ON-unit or procedure is entered during a GET or PUT operation, and within that ON-unit or procedure, a GET or PUT operation is executed for *x*, the value of COUNT is reset for the new operation. It is restored when the original GET or PUT is continued.

CURRENTSIZE

CURRENTSIZE returns a FIXED BIN(M,0) value giving the implementation-defined storage, in bytes, required by *x*.

The syntax for CURRENTSIZE is:



x a variable of any data type, data organization, and storage class except:

- A BASED, DEFINED, parameter, subscripted, or structure or union base-element variable that is an unaligned fixed-length bit string

- A minor structure or union whose first or last base element is an unaligned fixed-length bit string (except where it is also the first or last element of the containing major structure or union)
- A major structure or union that has the BASED, DEFINED, or parameter attribute, and which has an unaligned fixed-length bit string as its first or last element
- A variable not in connected storage.

The value returned by CURRENTSIZE(x) is defined as the number of bytes that would be transmitted in the following circumstances:

```
declare F file record output
        environment(scalarvarying);
write file(F) from(x);
```

If *x* is a scalar varying-length string, the returned value includes the length-prefix of the string and the number of currently-used bytes. It does not include any unused bytes in the string.

If *x* is a scalar area, the returned value includes the area control bytes and the current extent of the area. It does not include any unused bytes at the end of the area.

If *x* is an aggregate containing areas or varying-length strings, the returned value includes the area control bytes, the maximum sizes of the areas, the length prefixes of the strings, and the number of bytes in the maximum lengths of the strings. The exception to this rule is:

> If *x* is a structure or union whose last element is a nondimensioned area, the returned value includes that area's control bytes and the current extent of that area. It does not include any unused bytes at the end of that area.

For examples of the CURRENTSIZE built-in function, refer to the "SIZE" on page 436 and "MAXLENGTH" on page 413.

CURRENTSTORAGE

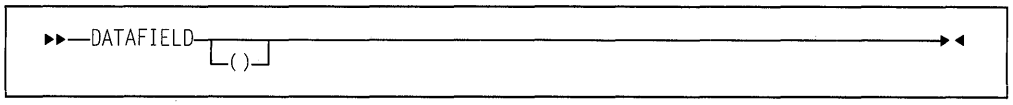
Abbreviation: CSTG

CURRENTSTORAGE is a synonym for CURRENTSIZE. For more information, refer to "CURRENTSIZE" on page 394.

DATAFIELD

DATAFIELD is in context in a NAME condition ON-unit (or any of its dynamic descendants). It returns a character string whose value is the contents of the field that raised the condition. It is also in context in an ON-unit (or any of its dynamic descendants) for an ERROR or FINISH condition raised as part of the implicit action for the NAME condition.

The syntax for DATAFIELD is:



If the string that raised the condition contains DBCS identifiers, GRAPHIC data, or mixed character data, DATAFIELD returns a mixed character string.

If DATAFIELD is used out of context, a null string is returned.

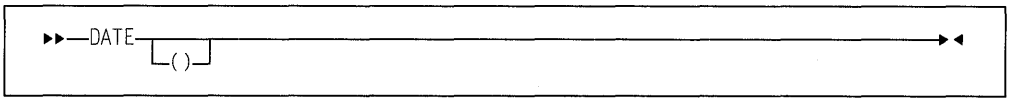
DATE

DATE returns a character string timestamp (length 6) with the format YYMMDD, in which:

YY is the current year
MM is the current month
DD is the current day

The time zone and accuracy are system dependent.

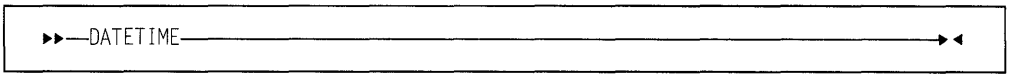
The syntax for DATE is:



DATETIME

DATETIME returns a character string timestamp of today's data in the default format.

The syntax for DATETIME is:



The format of the timestamp is 'YYYYMMDDHHMISS999', in which

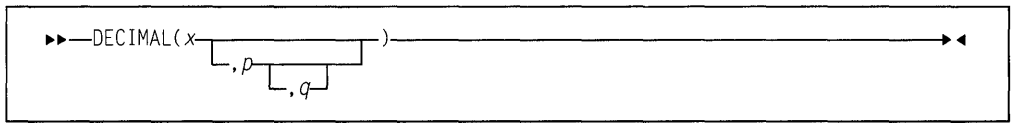
YYYY is the current year.
MM is the current month.
DD is the current day.
HH is the current hour.
MI is the current minute.
SS is the current second.
999 is the current millisecond.

If today's date is not available from the system, the ERROR condition is raised.

DECIMAL

DECIMAL returns the decimal value of *x*, with a precision specified by *p* and *q*. The result has the mode and scale of *x*.

The syntax for DECIMAL is:



Abbreviation: DEC

x reference.

p restricted expression specifying the number of digits to be maintained throughout the operation.

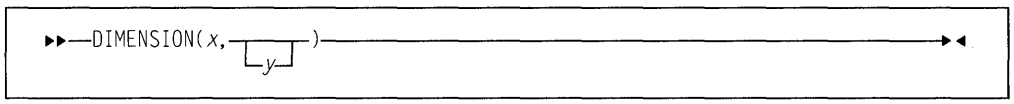
q restricted expression specifying the scaling factor of the result. For a fixed-point result, if *p* is given and *q* is omitted, a scaling factor of zero is assumed. For a floating-point result, *q* must be omitted.

If both *p* and *q* are omitted, the precision of the result is determined from the rules for base conversion.

DIMENSION

DIMENSION returns a FIXED BIN(M,0) value specifying the current extent of dimension *y* of *x*.

The syntax for DIMENSION is:



Abbreviation: DIM

x array reference. *x* must not have less than *y* dimensions.

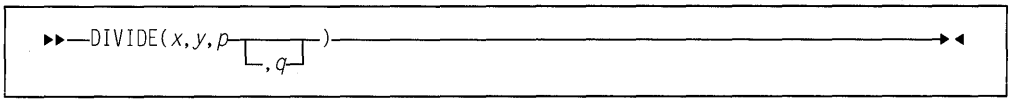
y expression specifying a particular dimension of *x*. If necessary, *y* is converted to a FIXED BIN(M,0). *y* must be greater than or equal to 1. If *y* is not supplied, it defaults to 1.

If *y* exceeds the number of dimensions of *x*, the DIMENSION function returns an undefined value.

DIVIDE

DIVIDE returns the quotient of *x/y* with a precision specified by *p* and *q*. The base, scale, and mode of the result follow the rules for expression evaluation.

The syntax for DIVIDE is:



x expression.

y expression. If $y = 0$, the ZERODIVIDE condition is raised.

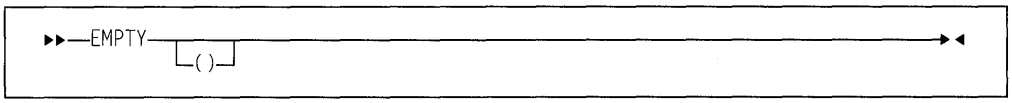
p restricted expression specifying the number of digits to be maintained throughout the operation.

q restricted expression specifying the scaling factor of the result. For a fixed-point result, if *q* is omitted, a scaling factor of zero is the default. For a floating-point result, *q* must be omitted.

EMPTY

EMPTY returns an area of zero extent. It can be used to free all allocations in an area.

The syntax for EMPTY is:



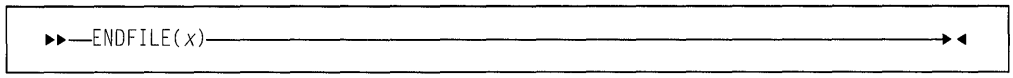
The value of this function is assigned to an area variable when the variable is allocated. For example:

```
declare A area,
        I based (P),
        J based (Q);

allocate I in(A), J in (A);
A = empty();

/* Equivalent to:  free I in (A), J in (A); */
```

ENDFILE

ENDFILE returns a '1'B when the end of the file is reached; '0'B if the end is not reached. If the file is not open, the ERROR condition is raised.

The syntax for ENDFILE is:

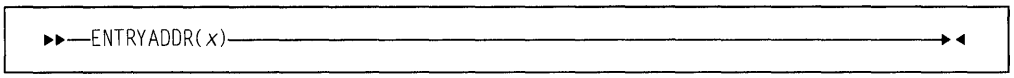


x file reference.

ENTRYADDR

ENTRYADDR returns a pointer value that is the address of the first executed instruction if the entry *x* is invoked. The entry *x* must represent a non-nested procedure.

The syntax for ENTRYADDR is:



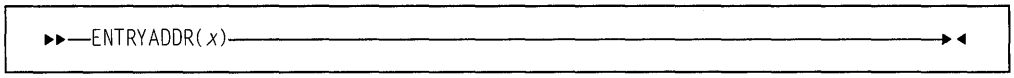
x entry reference.

If *x* is a fetchable entry constant, it must be fetched before ENTRYADDR is executed.

ENTRYADDR pseudovariable

The ENTRYADDR pseudovariable initializes an entry variable, *x*, with the address of the entry to be invoked.

The syntax for the ENTRYADDR pseudovariable is:



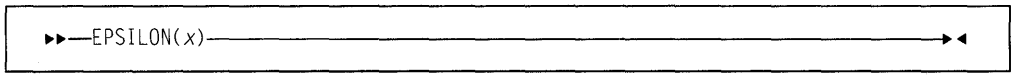
x entry reference.

Note: If the address supplied to the ENTRYADDR variable is the address of an internal procedure, the results are unpredictable.

EPSILON

EPSILON returns a floating-point value that is the spacing between x and the next positive number when x is 1. It has the base, mode, and precision of x.

The syntax for EPSILON is:



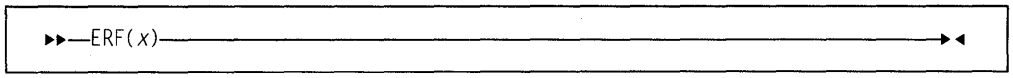
x expression declared as REAL FLOAT.

EPSILON(x) is a constant and may be used in restricted expressions.

ERF

ERF returns a real floating-point value that is an approximation of the error function of *x*.

The syntax for ERF is:



x real expression.

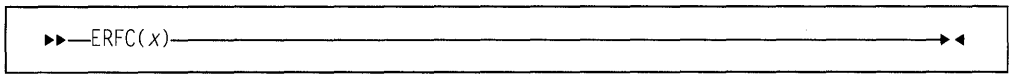
The result has the base and precision of *x*, and a value given by:

$$(2/function\sqrt{(\pi)}\,)\int_0^x \mathrm{EXP}(-(t^2))\,dt$$

ERFC

ERFC returns a real floating-point value that is an approximation of the complement of the error function of *x*.

The syntax for ERFC is:



x real expression.

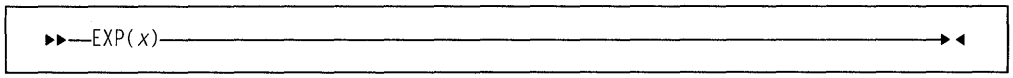
The result has the base and precision of *x*, and a value given by:

1 – ERF(*x*)

EXP

EXP returns a floating-point value that is an approximation of the base, e, of the natural logarithm system raised to the power *x*.

The syntax for EXP is:



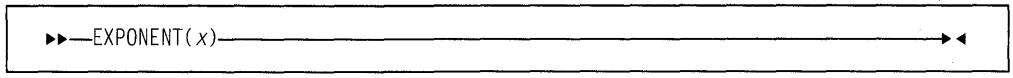
x expression.

The result has the base, mode, and precision of *x*.

EXPONENT

EXPONENT returns a FIXED BIN(M,0) value that is the exponent part of x.

The syntax for EXPONENT is:



x expression. *x* must be declared as REAL FLOAT.

EXPONENT(x) is not the "mathematical" exponent of *x*.
If *x* = 0, EXPONENT(x) = 0. For other values of *x*, EXPONENT(x) is the unique number *e* such that:

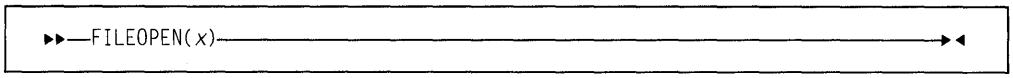
$$\text{radix}(x)^{(e-1)} \leq \text{abs}(x) < \text{radix}(x)^{e}$$

Consequently, EXPONENT(1e0) equals 1 and not 0.

FILEOPEN

FILEOPEN returns '1'B if the file *x* is open; '0'B if the file is not open.

The syntax for FILEOPEN is:

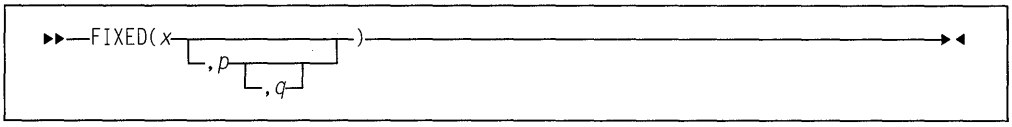


x file reference.

FIXED

FIXED returns the fixed-point value of *x*, with a precision specified by *p* and *q*. The result has the base and mode of *x*.

The syntax for FIXED is:



x expression.

p restricted expression that specifies the total number of digits in the result. It must not exceed the implementation limit.

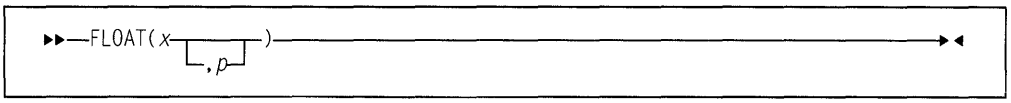
q restricted expression that specifies the scaling factor of the result. If *q* is omitted, a scaling factor of zero is assumed.

If both *p* and *q* are omitted, the precision of the result is determined from the rules for base conversion.

FLOAT

FLOAT returns the approximate floating-point value of *x*, with a precision specified by *p*. The result has the base and mode of *x*.

The syntax for FLOAT is:



x expression.

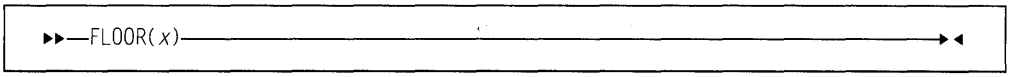
p restricted expression that specifies the minimum number of digits in the result.

If *p* is omitted, the precision of the result is determined from the rules for base conversion.

FLOOR

FLOOR determines the largest integer value less than or equal to *x*, and assigns this value to the result.

The syntax for FLOOR is:



x real expression.

The mode, base, scale and precision of the result match the argument. Except when *x* is fixed-point with precision (*p,q*), the precision of the result is given by:

```
(min(N,max(p-q+1,1)),0)
```

where N is the maximum number of digits allowed.

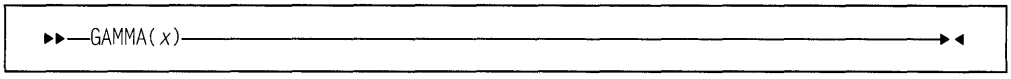
GAMMA

GAMMA is an approximation of the gamma of *x*:

$$\text{gamma}(x) = \int_0^{\infty} (u^{x-1})(e^{-x})du$$

GAMMA returns a floating-point value that has the base, mode, and precision of *x*.

The syntax for GAMMA is:



x expression. *x* must have a computational, arithmetic type. If *x* is numeric, it must be real and greater than zero. If *x* is not float, it will be converted to float.

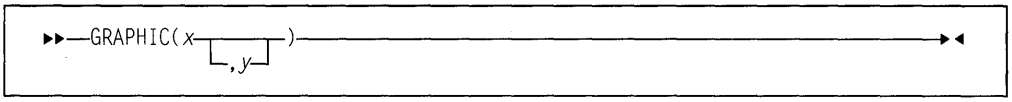
GRAPHIC

GRAPHIC can be used to explicitly convert character (or mixed character) data to GRAPHIC data. All other data first converts to character, and then to the GRAPHIC data type.

GRAPHIC returns the graphic value of *x*, with a length in graphic symbols specified by *y*.

Characters convert to graphics. The content of *x* is checked for validity during conversion, using the same rules as for checking graphic and mixed character constants.

The syntax for GRAPHIC is:



x expression. When *x* is GRAPHIC, it is subject to a length change, with applicable padding or truncation. When *x* is nongraphic, it is converted to character, if necessary. SBCS characters are converted to equivalent DBCS characters.

y expression. If necessary, *y* is converted to a real fixed-point binary value. If *y* is omitted, the length is determined by the rules for type conversion.

y must not be negative.

If *y* = 0, the result is the null graphic string.

If *y* is greater than the length needed to contain the graphic string, the result is padded with graphic blanks.

If *y* is less than the length needed to contain the graphic string, the result is truncated.

Example 1: Conversion from CHARACTER to GRAPHIC, where the target is long enough to contain the result:

```
dcl X char (11) varying;
dcl A graphic (11);
A = graphic(X,8);
```

For X with values	Intermediate Result	A is assigned
ABCDEFGHIJ	.A.B.C.D.E.F.G.H.I.J	.A.B.C.D.E.F.G.H.b.b.b
123	.1.2.3	.1.2.3.b.b.b.b.b.b.b
123A.B.C	.1.2.3.A.B.C	.1.2.3.A.B.C.b.b.b.b.b

where .b is a DBCS blank.

Example 2: Conversion from CHARACTER to GRAPHIC, where the target is too short to contain the result.

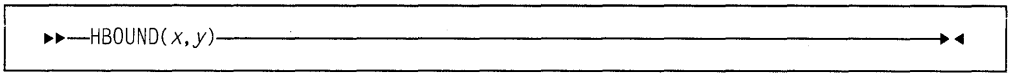
```
dcl X char (10) varying;
dcl A graphic (8);
A = graphic(X);
```

For X with Values	Intermediate Result	A Is Assigned
ABCDEFGHIJ	.A.B.C.D.E.F.G.H.I.J	.A.B.C.D.E.F.G.H

HBOUND

HBOUND returns a FIXED BIN(M,0) value specifying the current upper bound of dimension *y* of *x*.

The syntax for HBOUND is:



x array reference. *x* must not have less than *y* dimensions.

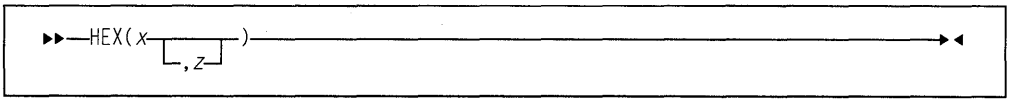
y expression specifying a particular dimension of *x*. If necessary, *y* is converted to FIXED BIN(M,0). *y* must be greater than or equal to 1. If *y* is not supplied, it defaults to 1.

HEX

HEX returns a character string that is the hexadecimal representation of the storage that contains *x*.

Note: This function does not return an exact image of *x* in storage. If an exact image is required, use the HEXIMAGE built-in function.

The syntax for HEX is:



HEX(x) returns a character string of length 2* size(*x*).

HEX(x,z) returns a character string that contains *x* with the character *z* inserted between every set of eight characters in the output string. Its length is 2* size(*x*) + ((size(*x*) – 1)/4).

x expression that represents any variable. The whole number of bytes that contain *x* are converted to hexadecimal.

z expression. If specified, *z* must result in CHARACTER(1) NONVARYING.

Example

```
dcl Sweet char(5) init('Sweet');
dcl Sixteen fixed bin(31) init(16);
dcl XSweet char(size(Sweet)*2+(size(Sweet)-1)/4);
dcl XSixteen char(size(Sixteen)*2+(size(Sixteen)-1)/4);

XSweet = hex(Sweet,'-');
          /* '53776565-74' */

XSweet = heximage(addr(Sweet),length(Sweet),'-');
          /* '53776565-74' */

XSixteen = hex(Sixteen,'-');
          /* '00000010' - bytes reversed */

XSixteen = heximage(addr(Sixteen),stg(Sixteen),'-');
          /* '10000000' - bytes NOT reversed */
```

HEXIMAGE

HEXIMAGE returns a character string that is the hexadecimal representation of the storage at a specified location.

The syntax for HEXIMAGE is:

HEXIMAGE(p,n) returns a character string that is the hexadecimal representation of n bytes of storage at location *p*. Its length is 2*n.

HEXIMAGE(p,n,z) returns a character string that is the hexadecimal representation of *n* bytes of storage at location *p* with character *z* inserted between every set of eight characters in the output string. Its length is (2*n) + ((n – 1)/4).

p restricted expression that must have a locator type (POINTER or OFFSET). If *p* is OFFSET, it must have the AREA attribute.

n expression. *n* must have a computational type and is converted to FIXED BIN(M,0).

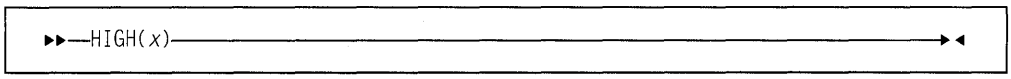
z If specified, *z* must result in CHARACTER(1) NONVARYING.

For examples of the HEXIMAGE built-in function, see "HEX" on page 404.

HIGH

HIGH returns a character string of length *x*, where each character is the highest character in the collating sequence (hexadecimal FF).

The syntax for HIGH is:



x expression. If necessary, *x* is converted to a positive real fixed-point binary value. If *x* = 0, the result is the null character string.

HUGE

HUGE returns a floating-point value that is the largest positive value *x* can assume. It has the base, mode, and precision of *x*.

The syntax for HUGE is:

x expression. *x* must have the attributes REAL FLOAT.

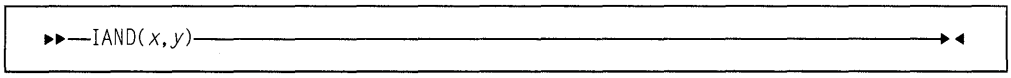
HUGE(x) is a constant and can be used in restricted expressions.

IAND

IAND returns the logical AND of *x* and *y*. The result of IAND(x,y) for *x* and *y* fixed is the value that a REAL FIXED BIN(M,0) variable *z* would have after the assignment:

unspec(*z*) = unspec(bin(*x*,31,0))&unspec(bin(*y*,31,0))

The syntax for IAND is:



x expression. *x* must have a computational type and is converted to FIXED BIN(M,0).

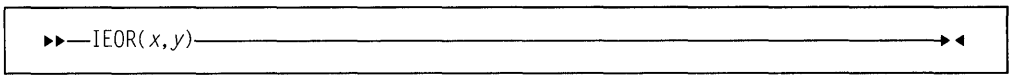
y expression. *y* must have a computational type and is converted to FIXED BIN(M,0).

IEOR

IEOR returns the logical exclusive-OR of *x* and *y*. The result of ieor(x,y) for *x* and *y* fixed is the value that a FIXED BIN(M,0) variable *z* has after the assignment:

unspec(*z*) = unspec(bin(*x*,31,0))¬unspec(bin(*y*,31,0))

The syntax for IEOR is:



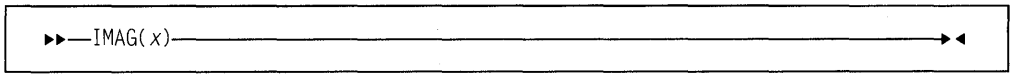
x expression. *x* must have a computational type and is converted to FIXED BIN(M,0).

y expression. *y* must have a computational type and is converted to FIXED BIN(M,0).

IMAG

IMAG returns the coefficient of the imaginary part of *x*. The mode of the result is real and has the base, scale, and precision of *x*.

The syntax for IMAG is:

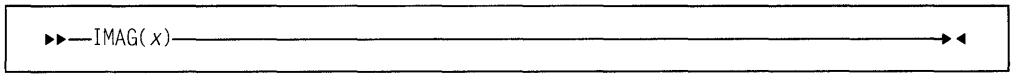


x expression. If *x* is real, it is converted to complex.

IMAG pseudovariable

The IMAG pseudovariable assigns a real value or the real part of a complex value to the coefficient of the imaginary part of *x*.

The syntax for IMAG is:

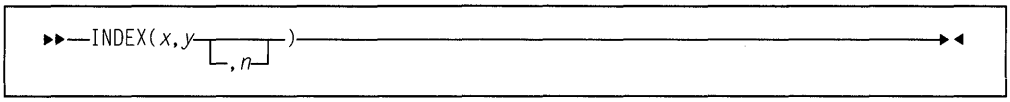


x complex reference.

INDEX

INDEX returns a real fixed-point binary value indicating the starting position within *x* of a substring identical to *y*. You can also specify the location within *x* where processing begins.

The syntax for INDEX is:



x string-expression to be searched.

y string-expression to be searched for.

n *n* specifies the location within *x* at which to begin processing. It must have a computational type and is converted to FIXED BIN(M,0).

If *y* does not occur in *x*, or if either *x* or *y* have zero length, the value zero is returned.

If *n* is less than 1 or if *n* is greater than 1 + length(x), the STRINGRANGE condition will be raised, and the result will be 0.

Example

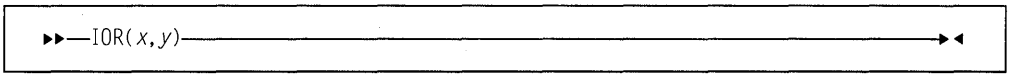
```
dcl greet char value ('Waheguru Ji Ka Khalsa, ' ||
                      'Waheguru Ji Ka Fateh');
dcl pos fixed bin init(1);

pos = index (greet, 'Ji', pos);
          /* 10              */

pos = index (greet, 'Ji', pos+1);
          /* 33              */

pos = index (greet, 'Ji', pos+1);
          /* 0               */
```

IOR

IOR returns the logical OR of *x* and *y*. The result of IOR(x,y) for *x* and *y* fixed is the value that a REAL FIXED BIN(M,0) variable *z* would have after the assignment:

unspec(*z*) = unspec(bin(*x*,31,0)) | unspec(bin(*y*,31,0)).

The syntax for IOR is:



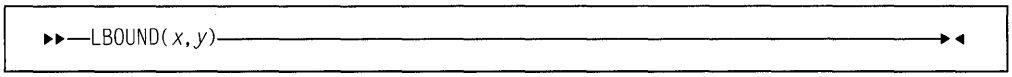
x expression. *x* must have a computational type and is converted to FIXED BIN(M,0).

y expression. *y* must have a computational type and is converted to FIXED BIN(M,0).

LBOUND

LBOUND returns a FIXED BINARY (M,0) value specifying the current lower bound of dimension *y* of *x*.

The syntax for LBOUND is:



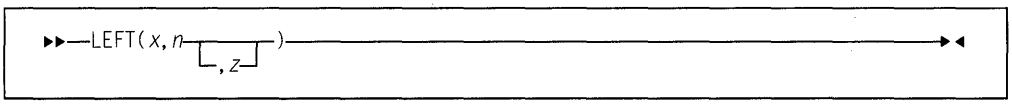
x array reference. *x* must not have less than *y* dimensions.

y expression specifying a particular dimension of *x*. If necessary, *y* is converted to FIXED BIN(M,0). *y* must be greater than or equal to 1. If *y* is not supplied, it defaults to 1.

LEFT

LEFT returns a string that is the result of inserting string *x* at the left end of a string with length *n* and padded on the right with the character *z* as needed. If *z* is omitted, a blank is used as the padding character.

The syntax for LEFT is:



x expression. *x* must have a computational type and should have a character type. If not, it is converted to CHARACTER.

n expression. *n* must have a computational type and should have a character type. If *n* does not have the attributes FIXED BIN(M,0), it is converted to them.

z expression. If specified, *z* must result in a CHARACTER(1) NONVARYING type.

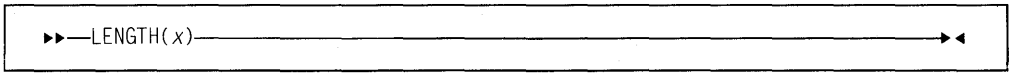
Example

```
dcl source char value('One Hundred Dollars');
dcl target char(25);

target = left (source, length(target), '*');
          /* 'One Hundred Dollars******'              */
```

LENGTH

LENGTH returns a real fixed-point binary value specifying the current length of *x*.

The syntax for LENGTH is:



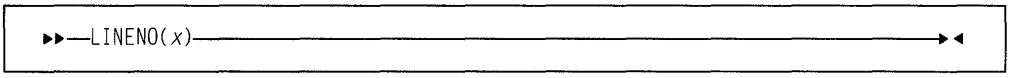
x string-expression. If *x* is binary, it is converted to bit string; otherwise, any other conversion required is to character string.

For examples of the LENGTH built-in function, refer to "SIZE" on page 436 and "MAXLENGTH" on page 413.

LINENO

LINENO returns a real fixed-point binary value specifying the current line number of *x*.

The syntax for LINENO is:



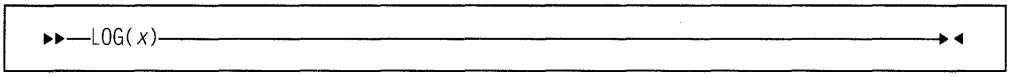
x file-reference.

The file must be open and have the PRINT attribute. If the file is not open or does not have the PRINT attribute, '0'B is returned.

LOG

LOG returns a floating-point value that is an approximation of the natural logarithm (the logarithm to the base e) of *x*. It has the base, mode, and precision of *x*.

The syntax for LOG is:



x expression. *x* must be greater than zero.

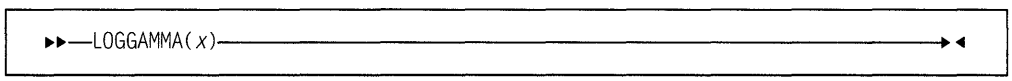
LOGGAMMA

LOGGAMMA returns a floating-point value that is an approximation of the log of gamma of *x*.

$$\text{GAMMA}(x) = \int_0^{\infty} (u^{(x-1)}) \exp(u)du$$

It has the base, mode, and precision of *x*.

The syntax for LOGGAMMA is:

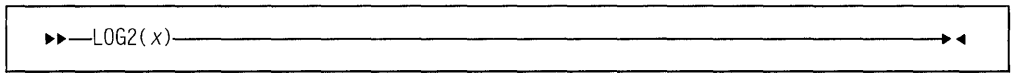


x expression. *x* must have a computational type and should have an arithmetic type. If *x* is numeric, it must be real and greater than zero. The expression *x* is converted to FLOAT.

LOG2

LOG2 returns a real floating-point value that is an approximation of the binary logarithm (the logarithm to the base 2) of *x*. It has the base and precision of *x*.

The syntax for LOG2 is:

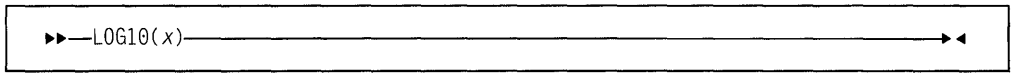


x real expression. The value of *x* must be greater than zero.

LOG10

LOG10 returns a real floating-point value that is an approximation of the common logarithm (the logarithm to the base 10) of *x*. It has the base and precision of *x*.

The syntax for LOG10 is:

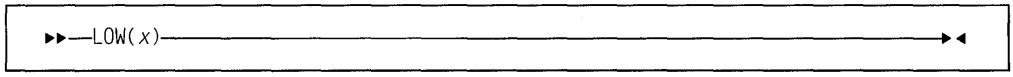


x real expression. It must be greater than zero.

LOW

LOW returns a character string of length *x*, where each character is the lowest character in the collating sequence (hexadecimal 00).

The syntax for LOW is:

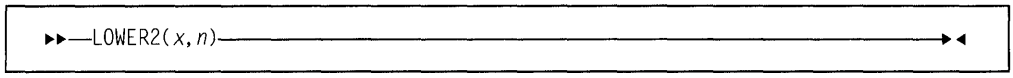


x expression. If necessary, *x* is converted to a positive real fixed-point binary value. If *x* = 0, the result is the null character string.

LOWER2

LOWER2(x,n) returns the value of floor($x*(2^{-n})$).

The syntax for LOWER2 is:



Note: LOWER2(x,n) is equivalent to the assembler SRA(x,n).

x and n
expressions. Both are converted to FIXED BIN(M,0).

The result is F!XED BIN(M,0). The result is undefined if *n* is negative or if *n* is greater than M.

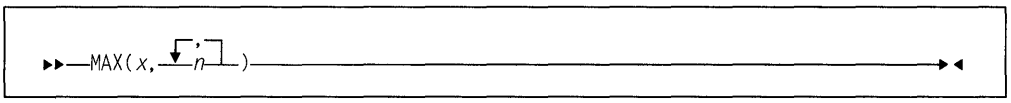
Examples

```
lower2 (+6,1)                    /*  Produces 3   */

lower2 (-6,1)                    /*  Produces -3  */

lower2 (-7,1)                    /*  Produces -4  */
```

MAX

MAX returns the largest value from a set of two or more expressions. When used in restricted expressions, exactly two arguments must be specified.

The syntax for MAX is:



x and n expressions.

All the arguments must be real. The result is real, with the common base and scale of the arguments.

If the arguments are fixed-point with precisions:

```
(p1,q1),(p2,q2),...,(pn,qn)
```

the precision of the result is given by:

```
(min(N,max(p1-q1,p2-q2,...,pn-qn) + max(q1,q2,...,qn)),max(q1,q2,...,qn))
```

where N is the maximum number of digits allowed.

If the arguments are floating point with precisions:

```
p1,p2,p3,...pn
```

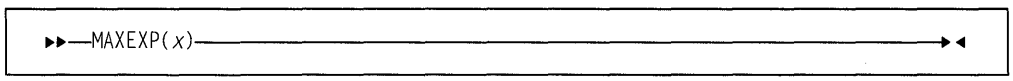
then the precision of the result is given by:

```
max(p1,p2,p3,...pn)
```

MAXEXP

MAXEXP returns a FIXED BIN(M,0) value that is the maximum value that EXPONENT(x) could assume.

The syntax for MAXEXP is:



x expression. *x* must have the REAL and FLOAT attributes.

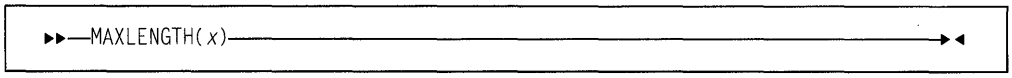
MAXEXP(x) is a constant and can be used in restricted expressions.

```
maxexp(x) = 128      for x float bin(p), p <= 21
maxexp(x) = 1024     for x float bin(p), 21 < p <= 53
maxexp(x) = 16384    for x float bin(p), 53 < p

maxexp(x) = 128      for x float dec(p), p <= 61
maxexp(x) = 1024     for x float dec(p), 6 < p <= 16
maxexp(x) = 16384    for x float dec(p), 16 < p
```

MAXLENGTH

MAXLENGTH returns the maximum length of a string.

The syntax for MAXLENGTH is:



x expression. *x* must have a computational type and can have a string type. If not, it is converted to character.

Example

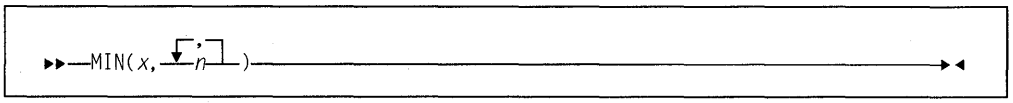
```
dcl scids   char            value('See you at SCIDS!');
dcl vscids  char(20) varying init('See you at SCIDS!');
dcl len fixed bin(31);

len = length (scids);          /* 17 characters */

len =    length (vscids);      /* 17 characters */

len = maxlength (vscids);      /* 20 characters */

len =    length (len);         /* 31 bits       */

len = maxlength (len);         /* 31 bits       */
```

For more examples, refer to "SIZE" on page 436.

MIN

MIN returns the smallest value from a set of two or more expressions. When used in restricted expressions, exactly two arguments must be specified.

The syntax for MIN is:



x and n expressions.

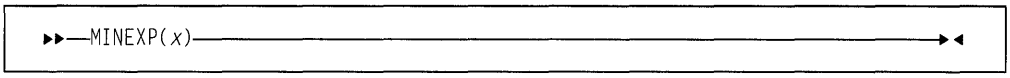
All the arguments must be real. The result is real with the common base and scale of the arguments.

The precision of the result is the same as that described in "MAX" on page 412.

MINEXP

MINEXP returns a FIXED BIN(M,0) value that is the minimum value that EXPONENT(x) could assume.

The syntax for MINEXP is:



x expression. *x* must have the REAL and FLOAT attributes

MINEXP(x) is a constant and can be used in restricted expressions.

```
minexp(x) = -125      for x float bin(p), p <= 21
minexp(x) = -1021     for x float bin(p), 21 < p <= 53
minexp(x) = -16381    for x float bin(p), 53 < p

minexp(x) = -125      for x float dec(p), p <= 6
minexp(x) = -1021     for x float dec(p), 6 < p <= 16
minexp(x) = -16381    for x float dec(p), 16 < p
```

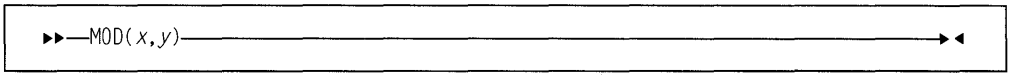
MOD

MOD returns the smallest nonnegative value, R, such that:

```
(x - R)/y = n
```

where *n* is an integer value. That is, R is the smallest nonnegative value that must be subtracted from *x* to make it divisible by *y*.

The syntax for MOD is:



x real expression.

y real expression. If *y* = 0, the ZERODIVIDE condition is raised.

The result, R, is real with the common base and scale of the arguments. If the result is floating-point, the precision is the greater of those of *x* and *y*. If the result is fixed-point, the precision is given by:

```
(min(N,p2-q2+max(q1,q2)),max(q1,q2))
```

where (p1,q1) and (p2,q2) are the precisions of *x* and *y*, respectively, and N is the maximum number of digits allowed.

If *x* and *y* are fixed-point with different scaling factors, the argument with the smaller scaling factor is converted to the larger scaling factor before R is calculated. If the conversion fails, the result is unpredictable.

The following example contrasts the MOD and REM built-in functions.

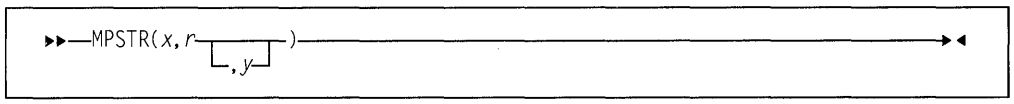
```
rem( +10, +8 ) = 2
mod( +10, +8 ) = 2

rem( +10, -8 ) = 2
mod( +10, -8 ) = 2

rem( -10, +8 ) = -2
mod( -10, +8 ) = 6

rem( -10, -8 ) = -2
mod( -10, -8 ) = 6
```

For information on the REM built-in function, see "REM" on page 430.

MPSTR

MPSTR truncates a string at a logical boundary and returns a mixed character string. For example, it does not truncate a double-byte character between bytes. The length of the returned string is equal to the length of the expression *x*, or to the value specified by *y*. The processing of the string is determined by the rules selected by the expression *r*, as described below.

The syntax for MPSTR is:



x expression that yields the character string result. The value of *x* cannot be GRAPHIC, and *x* is converted to character if necessary.

r expression that yields a character result. The expression cannot be GRAPHIC and is converted to character if necessary.

The expression *r* specifies the rules to be used for processing the string. The characters that can be used in *r* and the rules for them are as follows:

V or v Validates the mixed string *x* and returns a mixed string.

S or s Removes any null DBCS strings and creates a new string. Returns a mixed string.

If both V and S are specified, V takes precedence over S, regardless of the order in which they were specified.

If S is specified without V, the string x is assumed to be a valid string. If the string is not valid, undefined results occur.

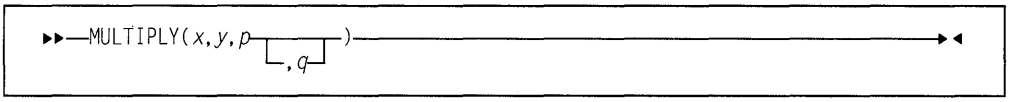
y expression. If necessary, *y* is converted to a real fixed-point binary value. If *y* is omitted, the length is determined by the rules for type conversion. The value of *y* cannot be negative. If $y = 0$, the result is the null character string. If *y* is greater than the length needed to contain *x*, the result is padded with blanks. If *y* is less than the length needed to contain *x*, the

result is truncated by discarding excess characters from the right (if they are SBCS characters), or by discarding as many DBCS characters (2-byte pairs) as needed.

MULTIPLY

MULTIPLY returns the product of *x* and *y*, with a precision specified by *p* and *q*. The base, scale, and mode of the result are determined by the rules for expression evaluation.

The syntax for MULTIPLY is:



x and y
expressions.

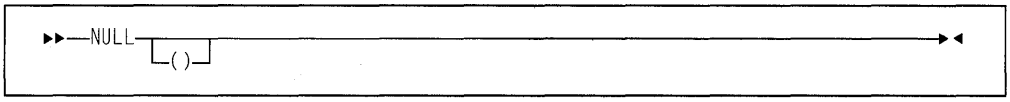
p restricted expression that specifies the number of digits to be maintained throughout the operation.

q restricted expression that specifies the scaling factor of the result. For a fixed-point result, if *q* is omitted, a scaling factor of zero is assumed. For a floating-point result, *q* must be omitted.

NULL

NULL returns the null pointer value. The null pointer value does not identify any generation of a variable. The null pointer value can be converted to OFFSET by assignment of the built-in function value to an offset variable.

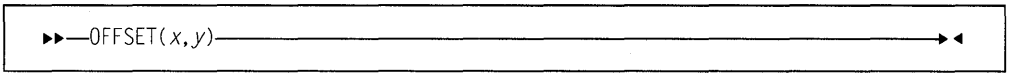
The syntax for NULL is:



OFFSET

OFFSET returns an offset value derived from a pointer reference *x* and relative to an area *y*. If x is the null pointer value, the null offset value is returned.

The syntax for OFFSET is:



x pointer reference. It must identify a generation of a based variable within the area *y*, or be the null pointer value.

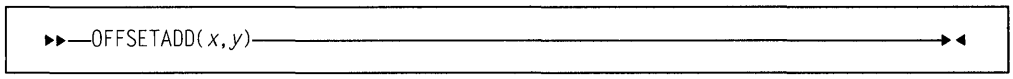
y area reference.

If *x* is an element reference, *y* must be an element variable.

OFFSETADD

OFFSETADD returns the sum of the arguments.

The syntax for OFFSETADD is:



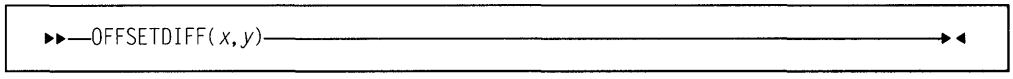
x expression. *x* must be specified as OFFSET.

y expression. *y* must have a computational type and is converted to FIXED BIN(M,0).

OFFSETDIFF

OFFSETDIFF returns a FIXED BIN(M,0) value that is the arithmetic difference between the arguments.

The syntax for OFFSETDIFF is:

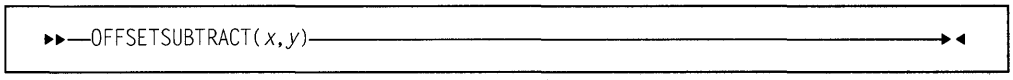


x and y
expressions. Both must be specified as OFFSET.

OFFSETSUBTRACT

OFFSETSUBTRACT is equivalent to OFFSETADD(x,-y).

The syntax for OFFSETSUBTRACT is:



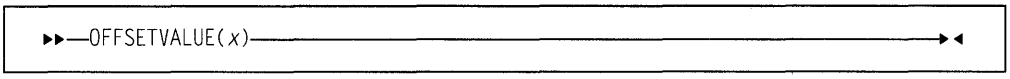
x expressions. *x* must be specified as OFFSET.

y expression. *y* must have a computational type and is converted to FIXED BIN(M,0)

OFFSETVALUE

OFFSETVALUE returns an offset value that is the converted value of *x*.

The syntax for OFFSETVALUE is:

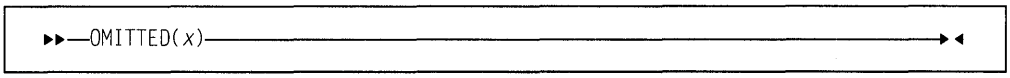


x expression. *x* must have a computational type and is converted to FIXED BIN(M,0).

OMITTED

OMITTED returns a BIT(1) value that is '1'B if the parameter named *x* was omitted in the invocation to its containing procedure.

The syntax for OMITTED is:

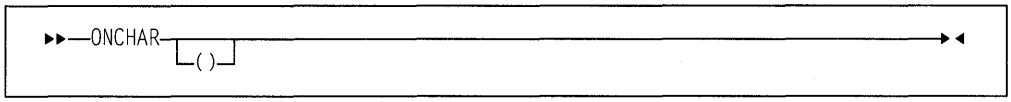


x level-one unscripted parameter with the BYADDR attribute.

ONCHAR

ONCHAR returns a 1-character string containing the character that caused the CONVERSION condition to be raised. It is in context in an ON-unit (or any of its dynamic descendants) for the CONVERSION condition or for the ERROR or FINISH condition raised as the implicit action for the CONVERSION condition.

The syntax for ONCHAR is:

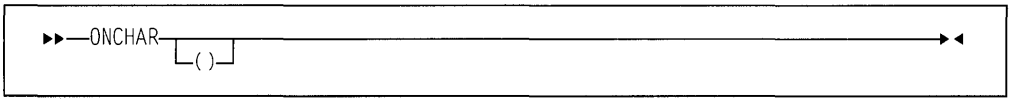


If the ONCHAR built-in function is used out of context, a blank is returned.

ONCHAR pseudovariable

The ONCHAR pseudovariable sets the current value of the ONCHAR built-in function. The element value assigned to the pseudovariable is converted to a character value of length 1. The new character is used when the conversion is re-attempted. (See "CONVERSION condition" on page 315.)

The syntax for ONCHAR pseudovariable is:



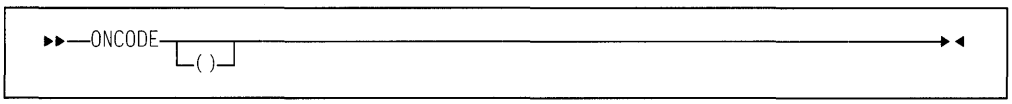
The pseudovariable must not be used out of context.

ONCODE

The ONCODE built-in function provides a fixed-point binary value that depends on the cause of the last condition. ONCODE can be used to distinguish between the various circumstances that raise a particular condition (for instance, the ERROR condition). For codes corresponding to the conditions and errors detected, see the specific condition in Chapter 16, "Conditions" on page 312.

ONCODE returns a real fixed-point binary value that is the condition code. It is in context in any ON-unit or its dynamic descendant. All condition codes are defined in Chapter 16, "Conditions" on page 312.

The syntax for ONCODE is:

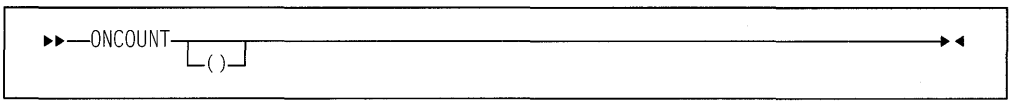


If ONCODE is used out of context, zero is returned.

ONCOUNT

ONCOUNT returns a FIXED BIN(M,0) value specifying the number of conditions that remain to be handled when an ON-unit is entered. Multiple conditions are discussed under "Multiple conditions" on page 309. It is in context in any ON-unit, or any dynamic descendant of an ON-unit.

The syntax for ONCOUNT is:



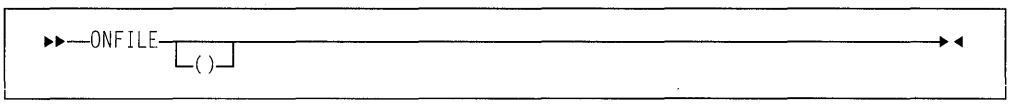
If ONCOUNT is used out of context, zero is returned.

ONFILE

ONFILE returns a character string whose value is the name of the file for which an input/output or CONVERSION condition is raised. If the name is a DBCS name, it will be returned as a mixed character string. It is in context in the following circumstances:

- In an ON-unit, or any of its dynamic descendants
- For any input/output or CONVERSION condition
- For the ERROR or FINISH condition raised as implicit action for an input/output or the CONVERSION condition.

The syntax for ONFILE is:

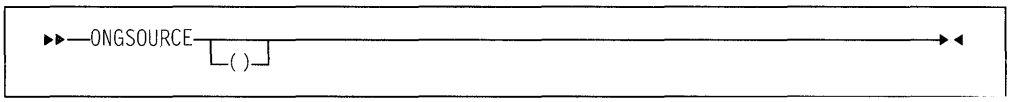


If ONFILE is used out of context, a null string is returned.

ONGSOURCE

ONGSOURCE returns a graphic string containing the DBCS character that caused the CONVERSION condition to be raised. It is in context in an ON-unit (or any of its dynamic descendants) for the CONVERSION condition or for the ERROR or FINISH condition raised as the implicit action for the CONVERSION condition.

The syntax for ONGSOURCE is:

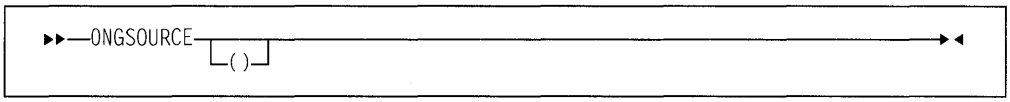


If the ONGSOURCE built-in function is used out of context, a DBCS blank is returned.

ONGSOURCE pseudovariable

The ONGSOURCE pseudovariable sets the current value of the ONGSOURCE built-in function. The element value assigned to the pseudovariable is converted graphic. The string is used when the conversion is re-attempted. (See "CONVERSION condition" on page 315.)

The syntax for ONGSOURCE pseudovariable is:



The pseudovariable must not be used out of context.

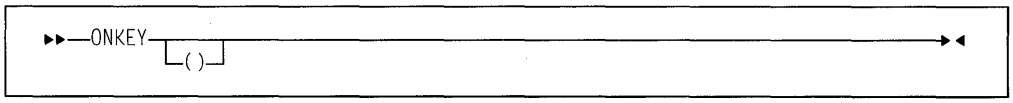
ONKEY

ONKEY returns a character string whose value is the key of the record that raised an input/output condition. For indexed files, if the key is GRAPHIC, the string is returned as a mixed character string. ONKEY is in context for the following:

- An ON-unit, or any of its dynamic descendants
- Any input/output condition, except ENDFILE
- The ERROR or FINISH condition raised as implicit action for an input/output condition.

ONKEY is always set for operations on a KEYED file, even if the statement that raised the condition has not specified the KEY, KEYTO, or KEYFROM options.

The syntax for ONKEY is:



The result of specifying ONKEY is:

- For any input/output condition (other than ENDFILE), or for the ERROR or FINISH condition raised as implicit action for these conditions, the result is the value of the recorded key from the I/O statement causing the error.
- For relative data sets, the result is a character string representation of the relative record number. If the key was incorrectly specified, the result is the last 8 characters of the source key. If the source key is less than 8 characters, it is padded on the right with blanks to make it 8 characters. If the key was correctly specified, the character string consists of the relative record number in character form padded on the left with blanks, if necessary.
- For a REWRITE statement that attempts to write an updated record on to an indexed data set when the key of the updated record differs from that of the input record, the result is the value of the embedded key of the input record.

If ONKEY is used out of context, a null string is returned.

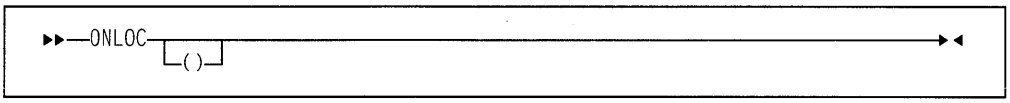
ONLOC

ONLOC returns a character string whose value is the name of the entry-point used for the current invocation of the procedure in which a condition was raised.

ONLOC always returns the first name of a multiple label specification, regardless of which name appears in the CALL or GOTO statement.

If the name is a DBCS name, it is returned as a mixed character string. It is in context in any ON-unit, or in any of its dynamic descendants.

The syntax for ONLOC is:

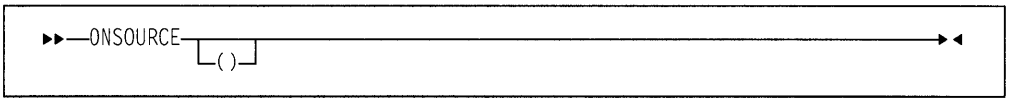


If ONLOC is used out of context, a null string is returned.

ONSOURCE

ONSOURCE returns a character string whose value is the contents of the field that was being processed when the CONVERSION condition was raised. It is in context in an ON-unit, or any of its dynamic descendants, for the CONVERSION condition or for the ERROR or FINISH condition raised as the implicit action for the CONVERSION condition.

The syntax for ONSOURCE is:

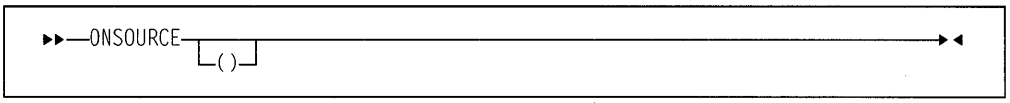


If ONSOURCE is used out of context, a null string is returned.

ONSOURCE pseudovariable

The pseudovariable sets the current value of the ONSOURCE built-in function. The element value assigned to the pseudovariable is converted to a character string and, if necessary, is padded on the right with blanks or truncated to match the length of the field that raised the CONVERSION condition. The string is used when the conversion is re-attempted. For information about conversion refer to "CONVERSION condition" on page 315.

The syntax for ONSOURCE pseudovariable is:



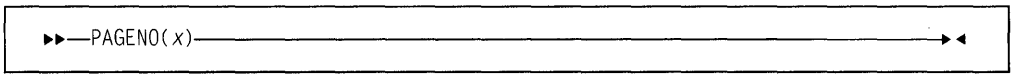
When conversion is re-attempted, the string assigned to the pseudovariable is processed as a single data item. For this reason, the error correction process must not assign a string containing more than one data item when the conversion occurs during the execution of a GET LIST or GET DATA statement. The presence of blanks or commas in the string could raise CONVERSION again.

The pseudovariable must not be used out of context.

PAGENO

PAGENO returns a FIXED BIN(M,0) value that is the current page number associated with file *x*.

The syntax for PAGENO is:



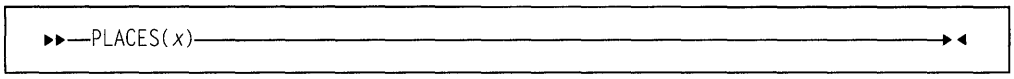
x an open PRINT file.

If the file is not a PRINT file, the ERROR condition is raised.

PLACES

PLACES returns a FIXED BIN(M,0) value that is the model-precision used to represent the floating point expression *x*.

The syntax for PLACES is:



x expression. *x* must be declared REAL FLOAT.

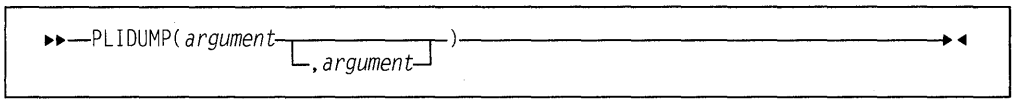
PLACES(x) is a constant and can be used in restricted expressions.

```
places(x) = 24        for x float bin(p), p <= 21
places(x) = 53        for x float bin(p), 21 < p <= 53
places(x) = 64        for x float bin(p), 53 < p

places(x) = 24        for x float dec(p), p <= 61
places(x) = 53        for x float dec(p), 6 < p <= 16
places(x) = 64        for x float dec(p), 16 < p
```

PLIDUMP

This built-in subroutine allows you to obtain a formatted dump of selected parts of storage used by your program.

The syntax for PLIDUMP is:

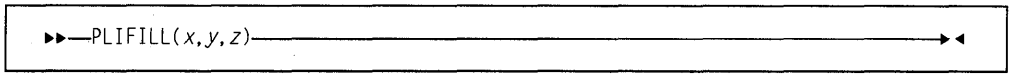


For more information about using PLIDUMP, see *PL/I Package/2 Programming Guide.*

PLIFILL

PLIFILL moves *z* copies of the byte *y* to the location *x* without any conversions, padding, or truncation.

The syntax for PLIFILL is:



x expression. *x* must be declared POINTER or OFFSET. If it is OFFSET, x must be declared with the AREA attribute.

y must be declared CHARACTER(1) NONVARYING.

z expression that is converted to FIXED BIN(M,0).

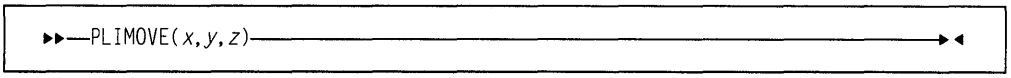
Example

```
dcl 1 str1,
      2 b  fixed bin(31),
      2 c  pointer,
      2 * union,
        3 d  char(4),
        3 e  fixed bin(31),
        3 *,
          4 * char(3),
          4 f fixed bin(8) unsigned,
      2 * char(0)
           initial call plifill( addr(str1), '00'x, stg(str1) );
```

PLIMOVE

PLIMOVE moves *z* storage units (bytes) from location *y* to location *x*, without any conversions, padding, or truncation.

The syntax for PLIMOVE is:



x and y
expressions declared as POINTER or OFFSET. If the type is OFFSET, *x* or *y* must be declared with the AREA attribute.

z expression. *z* must have a computational type and is converted to FIXED BIN(M,0).

Storage at locations *x* and *y* is assumed to be unique. If storage overlaps, unpredictable results can occur.

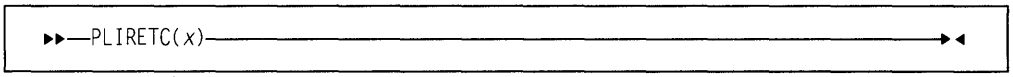
Example

```
dcl 1 str1,
      2 b  fixed bin(31),
      2 c  pointer,
      2 * union,
        3 d  char(4),
        3 e  fixed bin(31),
        3 *,
          4 * char(3),
          4 f fixed bin(8) unsigned,
      2 * char(0);
dcl 1 template nonasgn static,
      2 * fixed bin(31) init(200),
      2 * pointer init(null()),
      2 * char(4) init(''),
      2 * char(0);

call plimove(addr(str1), addr(template), stg(str1));
```

PLIRETC

This built-in subroutine allows you to set a return code that can be examined by the program that invoked this PL/I program or by another PL/I procedure via the PLIRETV built-in function.

The syntax for PLIRETC is:

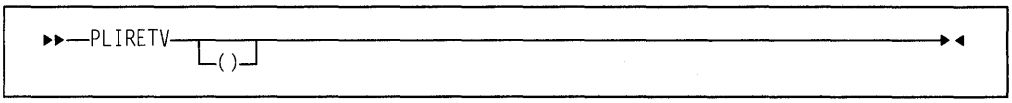


x an expression yielding a FIXED BINARY(M,0) return code.

PLIRETV

PLIRETV returns a FIXED BIN(M,0) value that is the PL/I return code.

The syntax for PLIRETV is:

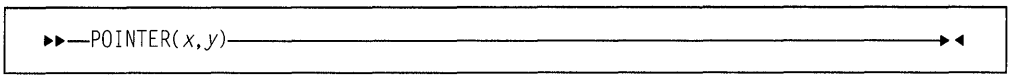


The value of the PL/I return code is the most recent value specified by a CALL PLIRETC statement.

POINTER

POINTER returns a pointer value that identifies the generation specified by an offset reference *x*, in an area specified by *y*. If *x* is the null offset value, the null pointer value is returned.

The syntax for POINTER is:



Abbreviation: PTR

x offset reference. It can be the null offset value. If it is not, *x* must identify a generation of a based variable, but not necessarily in *y*. If it is not in *y*, the generation must be equivalent to a generation in *y*.

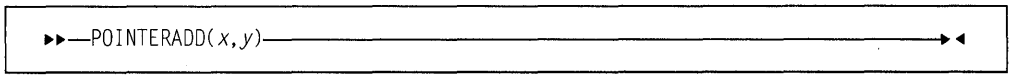
y area reference.

Generations of based variables in different areas are equivalent if, up to the allocation of the latest generation, the variables have been allocated and freed the same number of times as each other.

POINTERADD

POINTERADD returns a pointer value that is the sum of its arguments.

The syntax for POINTERADD is:



Abbreviation: PTRADD

x pointer expression.

y expression that must have a computational type and is converted to FIXED BIN(M,0).

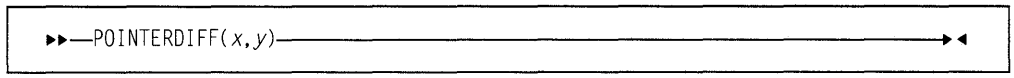
POINTERADD can be used as a locator for a based variable.

POINTERADD can be used for subtraction by prefixing the operand to be subtracted with a minus sign.

POINTERDIFF

POINTERDIFF returns a FIXED BIN(M,0) result that is the difference between the two pointers *x* and *y*.

The syntax for POINTERDIFF is:

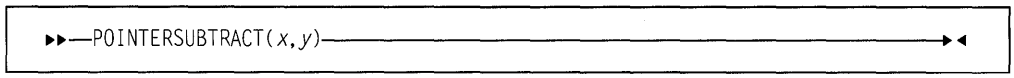


x and y
expressions declared as POINTER.

POINTERSUBTRACT

POINTERSUBTRACT is equivalent to POINTERADD(x,-y).

The syntax for POINTERSUBTRACT is:



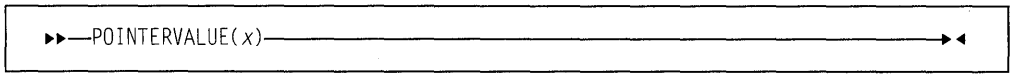
x must be a pointer expression.

y expression that must have a computational type and is converted to FIXED BIN(M,0).

POINTERVALUE

POINTERVALUE returns a pointer value that is the converted value of *x*.

The syntax for POINTERVALUE is:



Abbreviation: PTRVALUE

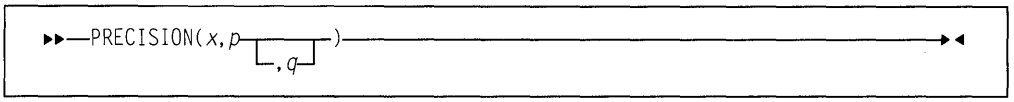
x expression that must have a computational type and is converted to FIXED BIN(M,0).

POINTERVALUE(x) can be used to initialize static pointer variables if *x* is a constant.

PRECISION

PRECISION returns the value of *x*, with a precision specified by *p* and *q*. The base, mode, and scale of the returned value are the same as that of *x*.

The syntax for PRECISION is:



Abbreviation: PREC

x expression.

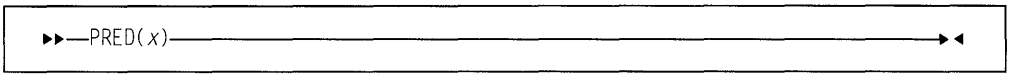
p restricted expression. *p* specifies the number of digits that the value of the expression *x* is to have after conversion.

q restricted expression. It specifies the scaling factor of the result. For a fixed-point result, if *q* is omitted, a scaling factor of zero is assumed. For a floating-point result, *q* must be omitted.

PRED

PRED returns a floating-point value that is the biggest representable number smaller than *x*. It has the base, mode, and precision of x. OVERFLOW will be raised if there is no such number.

The syntax for PRED is:

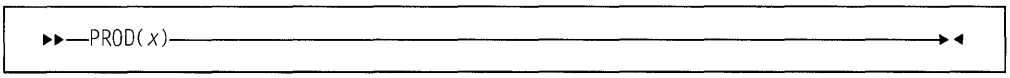


x expression declared REAL FLOAT.

PROD

PROD returns the product of all the elements in *x*.

The syntax for PROD is:



x array reference. If the elements of *x* are strings, they are converted to fixed-point integer values.

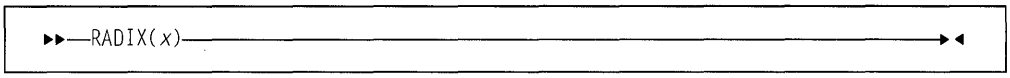
If the elements of *x* are not fixed-point integer values or strings, they are converted to floating-point and the result is floating-point.

The result has the precision of *x*, except that the result for fixed-point integer values and strings is fixed-point with precision (n,0), where *n* is the maximum number of digits allowed. The base and mode match the converted argument *x*.

RADIX

RADIX returns a FIXED BIN(M,0) value that is the model-base used to represent the floating point expression *x*.

The syntax for RADIX is:



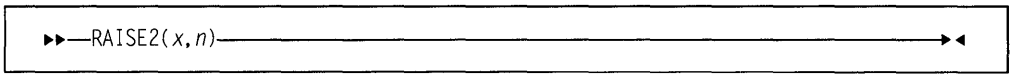
x expression declared REAL FLOAT.

RADIX(x) is 2 and can be used in restricted expressions.

RAISE2

RAISE2(x,n) returns the value of $x*(2^n)$.

The syntax for RAISE2 is:



Note: RAISE2(x,n) is equivalent to the assembler SLA(x,n).

x and n
expressions that must have a computational type and is converted to FIXED BIN(M,0).

The result has the type FIXED BIN(M,0). It is undefined if *n* is negative or if *n* is greater than M.

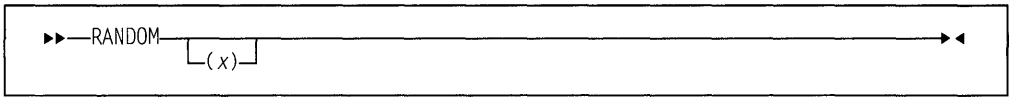
Example

```
raise2(6,1)                    /*  produces 12  */
```

RANDOM

RANDOM returns a FLOAT BIN(53) random number generated using *x* as the given seed. If *x* is omitted, the random number generated is based on the seed provided by the last RANDOM invocation with a seed, or on a default initial seed of 1 if RANDOM has not previously been invoked with a seed.

The syntax for RANDOM is:

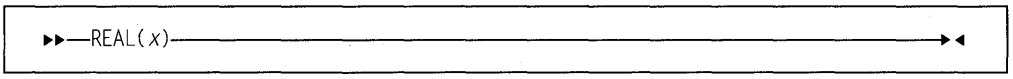


x expression. *x* must have a computational type and should have an arithmetic type. If *x* is numeric, it must be real. If *x* is not specified FIXED BIN(M,0), it is converted.

REAL

REAL returns the real part of *x*. The result has the base, scale, and precision of *x*.

The syntax for REAL is:

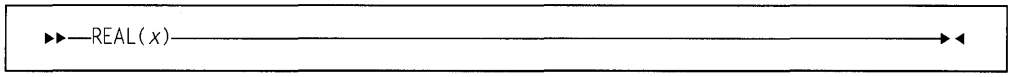


x expression. If *x* is real, it is converted to complex.

REAL pseudovariable

The pseudovariable assigns a real value or the real part of a complex value to the real part of *x*.

The syntax for REAL pseudovariable is:



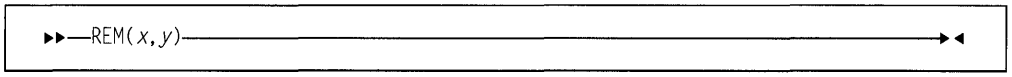
x complex reference.

REM

REM returns the remainder of *x* divided by *y*. This can be calculated by:

$x - y*\text{trunc}(x/y)$

The syntax for REM is:



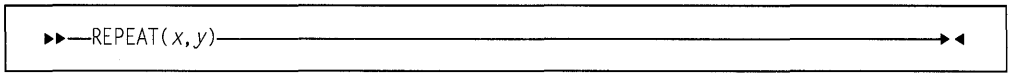
x and y
expressions. *x* and *y* must be computational and can be arithmetic.

For examples that contrast the REM and MOD built-in functions, refer to "MOD" on page 414.

REPEAT

REPEAT returns a bit or character string consisting of *x* concatenated to itself the number of times specified by *y*. That is, there will be (*y* + 1) occurrences of *x*.

The syntax for REPEAT is:



x bit- or character-expression to be repeated. If *x* is arithmetic, the following conversions occur:

- If it is binary, *x* is converted to bit string
- If it is decimal, *x* is converted to character string.

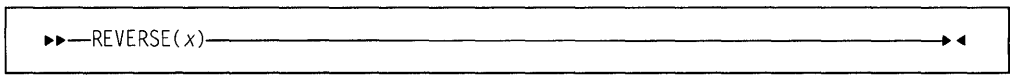
y expression. If necessary, *y* is converted to a real fixed-point binary value.

If *y* is zero or negative, the string *x* is returned. For an example of the REPEAT built-in function, see "COPY" on page 392.

REVERSE

REVERSE returns a nonvarying string that contains the elements of *x* in reverse order.

The syntax for REVERSE is:



x expression. *x* must have a computational type and should have a string type. If *x* does not have a string type, it is converted to string (that is, from numeric to bit or character), according to the rules for concatenation.

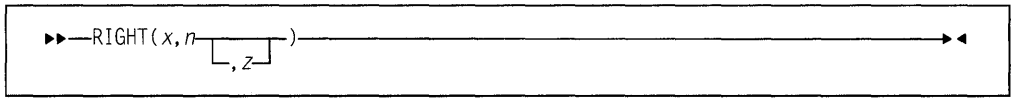
Example

```
dcl source char value('HARPO');
dcl target char(length(source));

target = reverse (source);      /* 'OPRAH' */
```

RIGHT

RIGHT returns a string that is the result of inserting string *x* at the right end of a string with length *n* and padded on the left with the character *z* as needed. If *z* is omitted, a blank is used as the padding character.

The syntax for RIGHT is:



x expression. *x* must have a computational type and can have a character type. If not, it is converted to character.

n expression that must have a computational type and is converted to FIXED BIN(M,0).

z expression. If specified, *z* must result in a CHARACTER(1) NON-VARYING type.

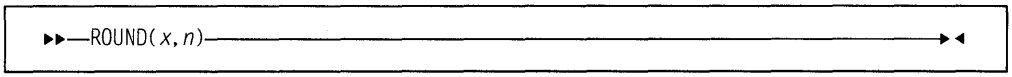
Example

```
dcl source char value('One Hundred Dollars');
dcl target char(25);

target = right (source, length(target), '*');
          /* '******One Hundred Dollars'              */
```

ROUND

The value of *x* is rounded at a digit specified by *n*. The result has the mode, base, and scale of *x*.

The syntax for ROUND is:



x real expression. If *x* is negative, the absolute value is rounded and the sign is restored.

n optionally-signed integer. It specifies the digit at which rounding is to occur. *n* must conform to the limits of scaling-factors for FIXED data. If *n* is greater than 0, rounding occurs at the (*n*)th digit to the right of the point. If *n* is zero or negative, rounding occurs at the (1-*n*)th digit to the left of the point.

The precision of a fixed-point result is given by:

```
(max(1,min(p-q+1+n,N)),n)
```

where (*p,q*) is the precision of *x*, and N is the maximum number of digits allowed. Thus, *n* specifies the scaling factor of the result.

In the following example:

```
dcl X fixed dec(5,4) init(6.6666);
put (round(X,2));
```

the value 6.67 is output.

If *x* is FIXED,

$$\mathtt{round}(x,n) = \mathtt{sign}(x)*(b^{-n})*\mathtt{floor}(\ \mathtt{abs}(x)*(b^{n-e}) + 1/2)$$

where b = 2 if *x* is BINARY; b = 10 if *x* is DECIMAL.

If *x* is FLOAT and not equal to 0,

$$\mathtt{round}(x,n) = \mathtt{sign}(x)*(b^{(e-n)})*\mathtt{floor}(\ \mathtt{abs}(x)*(b^{(e-n)}) + 1/2)$$

where b = radix(x) and e = exponent(x).

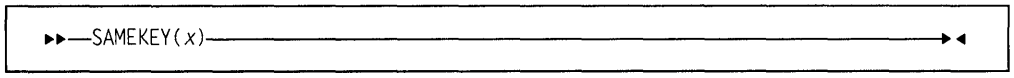
If *x* is FLOAT and equal to 0,

```
function round (x,n) = 0
```

SAMEKEY

SAMEKEY returns a bit string of length 1 indicating whether a record that has been accessed is followed by another with the same key.

The syntax for SAMEKEY is:



x file reference. The file must have the RECORD attribute.

Upon successful completion of an input/output operation on file *x*, or immediately before the RECORD condition is raised, the value accessed by SAMEKEY is set to '1'B if the record processed is followed by another record with the same key, and set to '0'B if it is not.

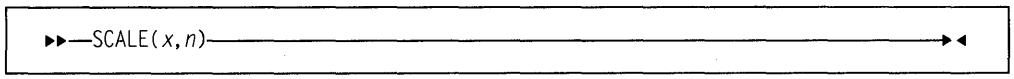
The value accessed by SAMEKEY is also set to '0'B if:

- An input/output operation that raises a condition other than RECORD also causes file positioning to be changed or lost
- The file is not open
- No current cursor position exists in the file.

SCALE

SCALE returns a floating-point value based on the formula $x*(\text{radix}(x)^n)$. The result has the base, mode, and precision of *x*.

The syntax for SCALE is:



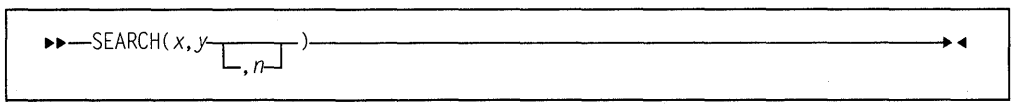
x expression declared REAL FLOAT.

n expression that must have a computational type and is converted to FIXED BIN(M,0).

SEARCH

SEARCH returns the first position in one string at which a character, bit, or graphic element of another string appears. It also allows you to specify the location at which to start searching.

The syntax for SEARCH is:



x and y

expressions. *x* specifies the string in which to search for any character, bit, or graphic in *y*.

n expression. *n* specifies the location within *x* at which to begin searching. It must have a computational type and is converted to FIXED BIN(M,0). If *n* is less than 1 or if *n* is greater than 1 + length(x), the STRINGRANGE condition is raised, and the result is zero.

Example

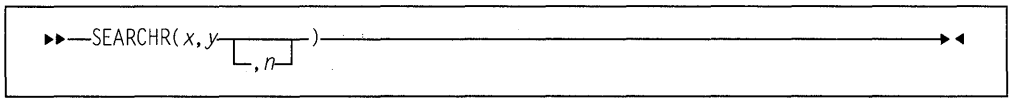
```
dcl source char value('  PL/I is the Power  ');
dcl (first, last) char(25) varying;
dcl wordlen fixed bin(31);
dcl pos fixed bin(31);
dcl start fixed bin(31);
dcl end fixed bin(31);

/* Find the first and the last word in "source", assuming      */
/* it contains at least one word                               */
start = verify(source, ' ');        /* find start of 1st word  */
end   = search(source, ' ',start+1);/* find end of word        */
if end = 0 then
   end = length(source)+1;
wordlen = end-start;
first  = substr(source,start,wordlen); /* 'PL/I'               */

end   = verifyr(source, ' ');       /* find end of last word   */
start = searchr(source, ' ',end-1); /* find start of word      */
wordlen = end-start;
last   = substr(source,start+1,wordlen); /* 'Power'            */
```

SEARCHR

The SEARCHR function performs the same operation as the SEARCH built-in function except that

- The search is done from right to left
- The default value for *n* is LENGTH(*x*).

The syntax for SEARCHR is:

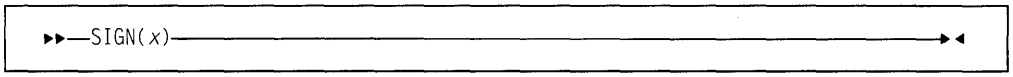


For argument descriptions and an example, refer to "SEARCH" on page 433.

SIGN

SIGN returns a FIXED BIN(M,0) value that indicates whether *x* is positive, zero, or negative.

The syntax for SIGN is:



x real expression.

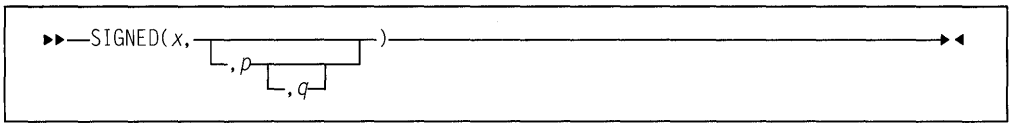
The returned value is given by:

Value of x	Value Returned
x > 0	+1
x = 0	0
x < 0	–1

SIGNED

SIGNED returns a signed FIXED BIN value of *x*, with a precision specified by *p* and *q*.

The syntax for SIGNED is:



x expression.

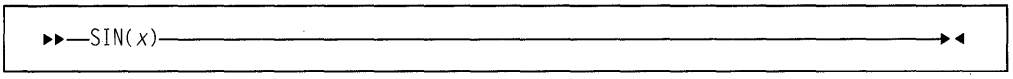
p restricted expression that specifies the number of digits to be maintained throughout the operation.

q restricted expression that specifies the scaling factor of the result. For a fixed-point result, if *p* is given and *q* is omitted, a scaling factor of zero is the default.

SIN

SIN returns a floating-point value that is an approximation of the sine of *x*. It has the base, mode, and precision of *x*.

The syntax for SIN is:

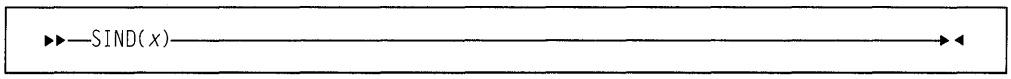


x expression whose value is in radians.

SIND

SIND returns a real floating-point value that is an approximation of the sine of *x*. It has the base and precision of *x*.

The syntax for SIND is:

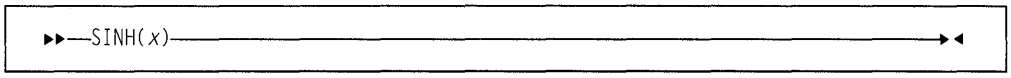


x real expression whose value is in degrees.

SINH

SINH returns a floating-point value that represents an approximation of the hyperbolic sine of *x*. It has the base, mode, and precision of *x*.

The syntax for SINH is:

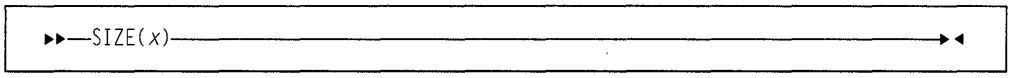


x expression whose value is in radians.

SIZE

SIZE returns a FIXED BIN(M,0) value giving the implementation-defined storage, in bytes, allocated to a variable *x*.

The syntax for SIZE is:



x a variable of any data type, data organization, alignment, and storage class, except as listed below.

x cannot be:

- A BASED, DEFINED, parameter, subscripted, or structure or union base-element variable that is an unaligned fixed-length bit string
- A minor structure or union whose first or last base element is an unaligned fixed-length bit string (except where it is also the first or last element of the containing major structure or union)
- A major structure or union that has the BASED, DEFINED, or parameter attribute, and which has an unaligned fixed-length bit string as its first or last element
- A variable not in connected storage.

The value returned by SIZE(x) is the maximum number of bytes that could be transmitted in the following circumstances:

```
declare F file record input
        environment(scalarvarying);
read file(F) into(x);
```

If *x* is:

- A varying-length string, the returned value includes the length-prefix of the string and the number of bytes in the maximum length of the string
- An area, the returned value includes the area control bytes and the maximum size of the area
- An aggregate containing areas or varying-length strings, the returned value includes the area control bytes, the maximum sizes of the areas, the length prefixes of the strings, and the number of bytes in the maximum lengths of the strings.

Example

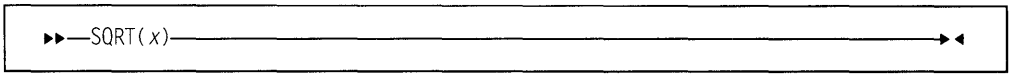
```
dcl scids   char(17)         init('See you at SCIDS!') static;
dcl vscids  char(20) varying init('See you at SCIDS!') static;
dcl stg fixed bin(31);

stg = storage   (scids);           /* 17 bytes */

stg = currentsize (scids);         /* 17 bytes */

stg = size (vscids);               /* 22 bytes */

stg = currentsize (vscids);        /* 19 bytes */

stg = size (stg);                  /* 4  bytes */

stg = currentsize (stg);           /* 4  bytes */
```

For other built-in functions that set extents, refer to "CURRENTSIZE" on page 394 and "MAXLENGTH" on page 413.

SQRT

SQRT returns a floating-point value that is an approximation of the positive square root of *x*. It has the base, mode, and precision of *x*.

The syntax for SQRT is:



x expression. If *x* is real, it must not be less than zero.

STORAGE

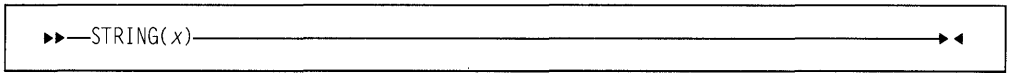
Abbreviation: STG

STORAGE is a synonym for SIZE.

STRING

STRING returns an element bit or character string that is the concatenation of all the elements of *x*.

The syntax for STRING is:



x aggregate or element reference.

STRING is restricted as follows:

- It cannot be applied to unions or structures containing unions
- If applied to a scalar, the scalar must be a nonvarying bit string, a nonvarying character string, a pictured character string, a pictured numeric string, or a nonvarying graphic string.
- If applied to a structure, the structure must contain no padding bytes and the elements of the structure must be either:
  - All unaligned nonvarying bit strings
  - All character strings, each of which is either a nonvarying character string, a pictured string, or a pictured numeric string.
  - All nonvarying graphic strings
- If applied to an array, the array must be connected and all the elements in the array are subject to the restrictions for STRING.

The following are valid STRING targets:

```
dcl
  1 A,
    2 B  bit(8),
    2 C  bit(2),
    2 D  bit(8);

dcl
  1 W,
    2 X  char(2),
    2 Y  pic'aa',
    2 Z  char(6);

dcl
  1 W,
    2 X  char(2),
    2 Y  pic'99',
    2 Z char(6);
```

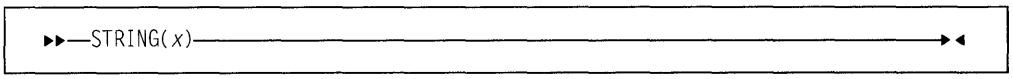
The following are invalid STRING targets:

```
dcl
  1 A,
    2 B  bit(8) aligned,
    2 C  bit(2),
    2 D  bit(8) aligned;
```

STRING pseudovariable

The STRING pseudovariable assigns a string to *x* as if *x* were a string scalar. Any remaining strings in *x* are filled with blanks or zero bits, or, if varying-length, are given zero length.

The syntax for STRING pseudovariable is:



x aggregate or element reference. Each base element of *x* must be either all bit-string or all character-string.

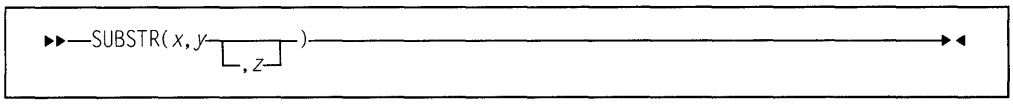
The STRING pseudovariable must not be used out of context.

The pseudovariable is also subject to the restrictions of the STRING built-in function. For more information on the restrictions, refer to 438.

SUBSTR

SUBSTR returns a substring, specified by *y* and *z*, of *x*.

The syntax for SUBSTR is:



x string expression. It specifies the string from which the substring is to be extracted. If *x* is not a string, it is converted to character.

y expression that is converted to FIXED BIN(M,0). *y* specifies the length of the substring in *x*.

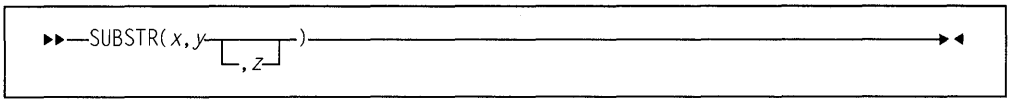
z expression that is converted to FIXED BIN(M,0). *z* specifies the length of the substring in *x*. If *z* is zero, a null string is returned. If *z* is omitted, the substring returned is position *y* in *x* to the end of *x*.

The STRINGRANGE condition is raised if *z* is negative or if the values of *y* and *z* are such that the substring does not lie entirely within the current length of *x*. It is not raised when *y* = LENGTH(*x*)+1 and *z* = 0. For an example of the SUBSTR built-in function, see "SEARCH" on page 433.

SUBSTR pseudovariable

The pseudovariable assigns a string value to a substring, specified by *y* and *z*, of *x*. The remainder of *x* is unchanged. (Assignments to a varying string do not change the length of the string.)

The syntax for SUBSTR pseudovariable is:



x string-reference. *x* must not be a numeric character.

y expression. *y* specifies the starting position of the substring in *x*. It can be converted to a real fixed-point binary value.

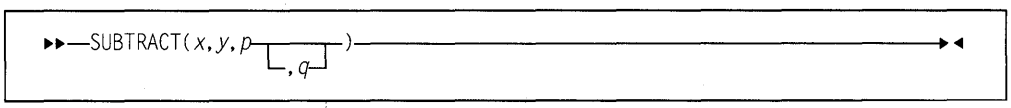
z expression. *z* specifies the length of the substring in *x*. it can be converted to a real fixed-point binary value. If *z* is zero, a null string is returned. If *z* is omitted, the substring returned is position *y* in *x* to the end of *x*.

y and *z* can be arrays only if *x* is an array.

SUBTRACT

SUBTRACT is equivalent to ADD(x,-y,p,q).

The syntax for SUBTRACT is:

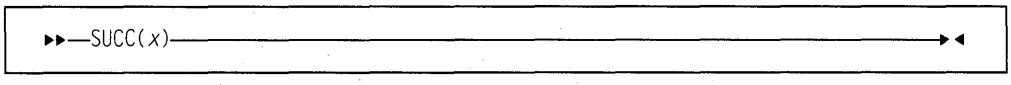


For details about arguments, refer to "ADD" on page 382 for argument descriptions.

SUCC

SUCC returns a floating-point value that is the smallest representable number larger than *x*. It is the base, mode, and precision of *x*. The OVERFLOW condition is raised if there is no such number.

The syntax for SUCC is:



x expression declared REAL FLOAT.

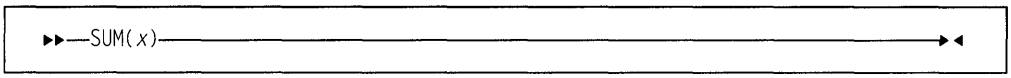
SUCC satisfies the following relationships:

```
pred(succ(x)) = x
succ(pred(x)) = x
succ(x)       = -pred(-x)
succ(0d0)     = tiny(0d0)
```

SUM

SUM returns the sum of all the elements in *x*. The base, mode, and scale of the result match those of *x*.

The syntax for SUM is:



x array expression. If the elements of *x* are strings, they are converted to fixed-point integer values.

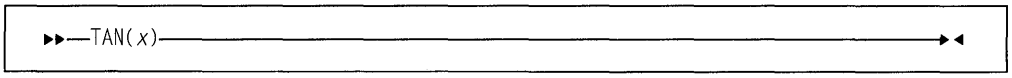
If the elements of *x* are fixed-point, the precision of the result is (*N,q*), where N is the maximum number of digits allowed, and *q* is the scaling factor of *x*.

If the elements of *x* are floating-point, the precision of the result matches *x*.

TAN

TAN returns a floating-point value that is an approximation of the tangent of *x*. It has the base, mode, and precision of *x*.

The syntax for TAN is:

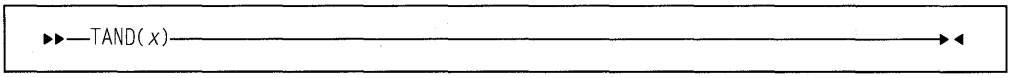


x expression whose value is in radians.

TAND

TAND returns a real floating-point value that is an approximation of the tangent of *x*. It has the base and precision of *x*.

The syntax for TAND is:

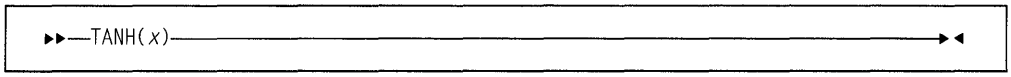


x real expression whose value is in degrees.

TANH

TANH returns a floating-point value that is an approximation of the hyperbolic tangent of *x*. It has the base, mode, and precision of *x*.

The syntax for TANH is:



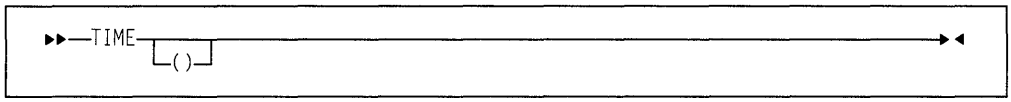
x expression whose value is in radians.

TIME

TIME returns a character string timestamp in the format HHMMSSTTT, in which:

HH current hour
MM current minute
SS current second

The syntax for TIME is:

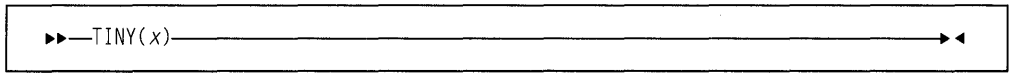


The time zone and accuracy are system dependent.

TINY

TINY returns a floating-point value that is the smallest positive value *x* can assume. it has the base, mode, and precision, of *x*.

The syntax for TINY is:



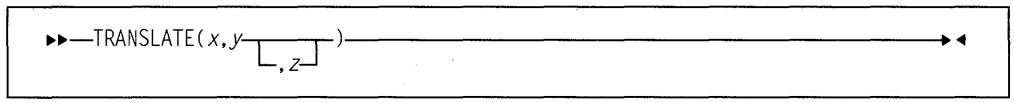
x expression declared REAL FLOAT.

Tiny(x) is a constant and can be used in restricted expressions.

TRANSLATE

TRANSLATE returns a character string of the same length as *x*.

The syntax for TRANSLATE is:



x character expression to be searched for possible translation of its characters.

y character expression containing the translation values of characters.

z character expression containing the characters that are to be translated. If *z* is omitted, it defaults to collate().

TRANSLATE operates on each character of *x* as follows:

If a character in *x* is found in *z*, the character in *y* that corresponds to that in *z* is copied to the result; otherwise, the character in *x* is copied directly to the result. If *z* contains duplicates, the leftmost occurrence is used.

y is padded with blanks, or truncated, on the right to match the length of *z*.

Any arithmetic or bit arguments are converted to character. TRANSLATE does not support GRAPHIC data.

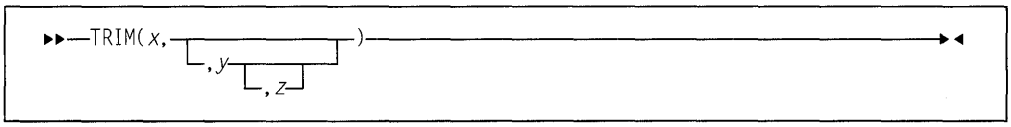
Example

```
dcl source char value("If it's Tuesday, it must be Deutsche abend!")
dcl target char(length(source));
dcl (to   value ('ABCDEFGHIJKLMNOPQRSTUVWXYZ'),
     from value ('abcdefghijklmnopqrstuvwxyz')) char(26);

target = translate(source, to, from);
         /* "IF IT'S TUESDAY, IT MUST BE DEUTSCHE ABEND!" */
```

TRIM

TRIM returns a nonvarying character string with characters trimmed from one or both ends.

The syntax for TRIM is:



x, y, and z

expressions.

Each must have a computational type and should have the attribute CHARACTER. If not, they are converted.

x is the string from which the characters defined by *y* are trimmed from the left, and the characters defined by *z* are trimmed from the right.

If *z* is omitted, it defaults to a CHARACTER(1) NONVARYING string containing one blank.

If *y* and *z* are both omitted, they both default to a CHAR(1) NONVARYING string containing one blank.

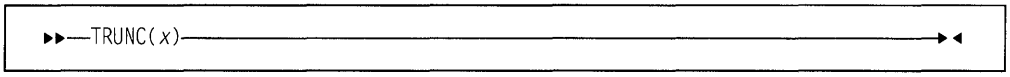
Example

```
dcl source char value(" *** PL/I's got the Power!  ***  ");
dcl target char(length(source)) varying;

target = trim(source, ' ', '* ');
            /* "*** PL/I's got the Power!" */
```

TRUNC

TRUNC returns an integer value that is the truncated value of *x*. If *x* is positive or 0, this is the largest integer value less than or equal to *x*. If *x* is negative, this is the smallest integer value greater than or equal to *x*. This value is assigned to the result.

The syntax for TRUNC is:



x real expression.

The base, mode, scale, and precision of the result match those of *x*. Except when x is fixed-point with precision (*p*,*q*), the precision of the result is given by:

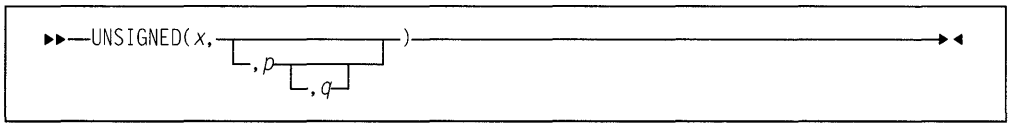
```
(min(N,max(p-q+1,1)),0)
```

where N is the maximum number of digits allowed.

UNSIGNED

UNSIGNED returns an unsigned FIXED BINARY value of *x*, with a precision specified by *p* and *q*.

The syntax for UNSIGNED is:



x expression.

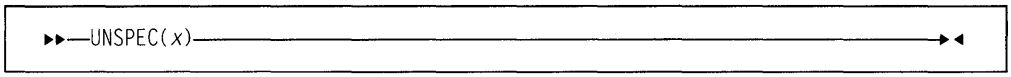
p integer. It specifies the number of digits to be maintained throughout the operation.

q optionally-signed integer. It specifies the scaling factor of the result. For a fixed-point result, if *p* is given and *q* is omitted, a scaling factor of zero is the default.

UNSPEC

UNSPEC returns a bit string that is the internal coded form of *x*.

The syntax for UNSPEC is:



x scalar, array, structure, or union expression.

The UNSPEC built-in function is subject to the following rules:

- For aggregates, UNSPEC is allowed only for those that contain no padding bytes or bits.
- The result will always be BIT(*) scalar. UNSPEC of an array does not yield an array of BIT(*).

Note: Use of UNSPEC can affect the portability of your application program.

The length of the returned bit string depends on the attributes of *x*, as shown in Figure 69.

Figure 69 (Page 1 of 2). Length of bit string returned by UNSPEC

Bit-String length	Attribute of *x*
8	SIGNED FIXED BIN (p,q), 1 <= p <= 7 UNSIGNED FIXED BIN (p,q), 1 <= p <= 8
16	SIGNED FIXED BINARY (p,q), 8 <= p <= 15 UNSIGNED FIXED BINARY (p,q), 9 <= p <= 16
32	ENTRY LIMITED SIGNED FIXED BINARY (p,q), 16 <= p <= 31 UNSIGNED FIXED BINARY (p,q), 17 <= p <= 31 FLOAT DECIMAL (p), 1 <= p <= 6 FLOAT BINARY (p), 1 <= p <= 21 OFFSET[2] FILE constant or variable POINTER
64	FLOAT BINARY(p), 21 < p < 53 FLOAT DECIMAL(p), 7 <= p <= 16 LABEL constant or variable ENTRY constant or variable
128	FLOAT BINARY(p), 54 <= p <= 109 FLOAT DECIMAL(p), 17 <= p <= 33
n	BIT (n)
8*n or 32767	CHARACTER (n) PICTURE (with character-string-value length of *n*) (when n>4096, a length of 32767 is returned)
16*n	GRAPHIC (n)
16+n	BIT VARYING where *n* is the maximum length of *x*
16+(8*n)	CHARACTER VARYING where *n* is the maximum length of *x*

Figure 69 (Page 2 of 2). Length of bit string returned by UNSPEC

Bit-String length	Attribute of *x*
16+(16*n)	GRAPHIC VARYING where *n* is the maximum length of *x*
8*(n+16)	AREA (n)[2]
8*FLOOR(n)	FIXED DECIMAL (p,q) where n = (p+2)/2

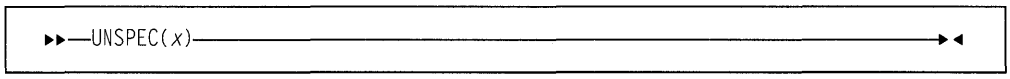
Alignment and storage requirements for program control data can vary across supported systems.

If *x* is a varying-length string, its two-byte prefix is included in the returned bit string. If *x* is an area, the returned value includes the control information.

UNSPEC pseudovariable

The UNSPEC pseudovariable assigns a bit value directly to *x*; that is, without conversion. The bit value is padded, if necessary, on the right with '0'B to match the length of *x*, according to Figure 69.

The syntax for the UNSPEC pseudovariable is:



x reference.

If *x* is a varying length string, its 2-byte prefix is included in the field to which the bit value is assigned. If *x* is an area, its control information is included in the receiving field.

The pseudovariable is subject to the rules for the UNSPEC built-in function described in "UNSPEC" on page 445.

Example

```
dcl 1 str1 nonasgn static,
      2 * fixed bin(15) init(16),  /* '1000'X */
      2 * char init('33'x),
      2 * bit init('1'b),
      2 ba(4) bit init('1'b, '0'b, '1'b, '0'b),
      2 b3 bit(3) init('111'b),
      2 * char(0);
dcl bit_str1 bit(size(str1)*8);
dcl bit_ba   bit(dim(ba)*length(ba(1)));
dcl bit_b3   bit(length(b3));

bit_ba   = unspec(ba);   /* result is scalar '1010'B not an array */
bit_b3   = unspec(b3);   /* '111'B                                */
bit_str1 = unspec(str1); /* '100033D7'B4 or
                            '100033'B4 || '11010111'B            */
```

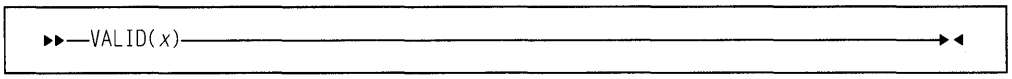
VALID

VALID returns a BIT(1) value that is '1'B under the following conditions:

- If *x* is PICTURE and its contents are valid for *x*'s picture specification.
- If *x* is FIXED DECIMAL and the data in *x* is valid fixed decimal data.

If these conditions are not met, the result is '0'B.

The syntax for VALID is:



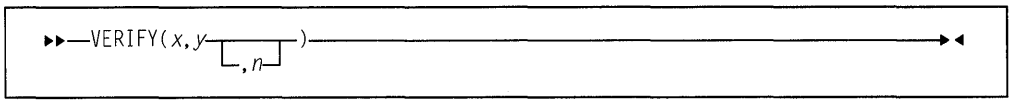
x reference with either picture or fixed decimal type.

VERIFY

VERIFY returns a real fixed-point binary value indicating the position in *x* of the leftmost character or bit that is not in *y*. It also allows you to specify the location within *x* at which to begin processing.

If all the characters or bits in *x* do appear in *y*, a value of zero is returned. If *x* is the null string, a value of zero is returned. If *x* is not the null string and *y* is the null string, a value of one is returned.

The syntax for VERIFY is:



x string-expression.

y string-expression.

n expression. *n* specifies the location within *x* where processing begins. It must have a computational type and is converted to FIXED BIN(M,0).

The STRINGRANGE condition, if enabled, is raised if 1 > *n* > length(*x*) + 1. Its implicit action and normal return give a result of 0. Refer to "Results under DEFAULT(ANS)" on page 64 and "Results under DEFAULT(IBM)" on page 63 for the results of any necessary conversions.

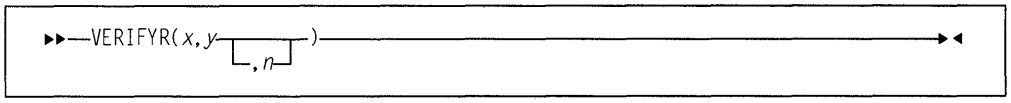
For an example of the VERIFY built-in function, see "SEARCH" on page 433.

VERIFYR

The VERIFYR function performs the same operation as the VERIFY built-in function except that

- The verification is done from right to left
- The default value for *n* is LENGTH(*x*).

The syntax for VERIFYR is:



For argument descriptions, refer to "VERIFY" on page 447. For an example, see "SEARCH" on page 433.

Chapter 18. Macro facility

Chapter 18. Macro facility

The PL/I Package/2 compiler provides a macro facility for source program alteration and conditional compilation. It is executed prior to compilation when you specify the MACRO compiler option. The macro facility scans the input source text and produces potentially modified output source text. The output can serve as input to the compiler or another source processor.

The description of the macro facility assumes that you know the PL/I language described in this publication. Terms, syntax, and semantics are consistent with PL/I rules unless otherwise noted.

Input source text is a stream of characters consisting of:

- *Macro facility statements.*

  Macro facility statements are executed as they are encountered during the scan. Macro facility statements and clauses begin with a percent symbol (%).

  The macro facility executes these statements and alters the source text accordingly. Macro facility statements can cause alteration of the source text in any of the following ways:

  - Any identifier appearing in the source text can be changed to arbitrary text.
  - Portions of the source text can be conditionally selected as the output source text.
  - Source files can be included into the source text.
- *Control statements.* These direct compilation process.
- *Source text.* Text that is not one of the above.

Macro facility output [1] is a stream of characters consisting of:

- *Control statements.* Control statements that are processed and generally copied to the output source text.
- *Source text.* Possibly altered source.

You can specify compiler options for printing the input source text or writing the output source text to a file. Compiler options are described in *PL/I Package/2 Programming Guide.*

Macro facility scan

The macro facility starts its scan at the beginning of the source input and scans each character sequentially. Scanning proceeds as follows, for:

Macro facility statements: These statements are executed when encountered. You can:

- Declare identifiers using the %DECLARE statement and make them eligible for replacement.

[1] Macro facility replacement output is shown in a formatted style, while the actual output is unformatted.

- Change values of macro facility variables using the %assignment statement.
- Make an identifier eligible for replacement, by its value, in the source text using the %ACTIVATE statement.
- Make an identifier ineligible for replacement, by its value, in the source text using the %DEACTIVATE statement.
- Generate a message using the %NOTE statement.
- Include a file using %INCLUDE.
- Cause the macro facility to continue the scan at a different point in the source input using the %GO TO or %IF statement.

Control statements: Control statements are generally copied into the macro facility output. A control statement must be on a line by itself.

Replacing identifiers in source text: The source text, after replacement of any active identifiers by new values, is copied into the output text. The source text is scanned for:

- Characters in the PL/I character set. Characters that are outside of the PL/I character set and are not part of constants and comments are treated as delimiters and are copied unchanged to the output text.
- PL/I constants or PL/I comments. These are copied unchanged to output text.
- Active Identifiers. For an identifier to be replaced its value, the identifier must be *active*. A %DECLARE statement implicitly activates the identifier being declared. It can be deactivated by a %DEACTIVATE statement and reactivated by an %ACTIVATE statement.

  An identifier in the source text that matches the name of an active macro facility identifier is replaced in the output text by the value of that identifier.

  Identifiers can be activated with either the RESCAN or the NORESCAN options. If the NORESCAN option applies, the value is immediately placed into the output text. If the RESCAN option applies, the replaced value is rescanned to determine whether it contains any active identifiers that should be replaced. This rescanning and replacement activity continues until no further replacements can be made, at which time the replacement text is placed into the output text. Thus, insertion of a value into the output text takes place only after all possible replacements have been made.

Replacement values must not contain % symbols, unmatched comments, or unmatched single or double quotation marks that enclose constants.

The scan terminates when all source input has been processed. The macro facility output is then complete for processing by the compiler or another processor.

Character sets

The macro facility supports only the single-byte character set, as defined in "Single-byte character set" on page 10.

Reserved keywords

The macro facility reserves some keywords that have specific meaning to it. Reserved keywords cannot be used as identifier names.

Macro facility keywords are shown in Figure 70.

Figure 70. Macro facility keywords

Keyword	Abbreviation	Status
ACTIVATE	ACT	Reserved
ANSWER	ANS	
BY		
CHARACTER	CHAR	Unreserved
CODE		
COLUMN	COL	
DEACTIVATE	DEACT	Reserved
DECLARE	DCL	Reserved
DO		Reserved
ELSE		Reserved
END		Reserved
EXTERNAL	EXT	Unreserved
FIXED		Unreserved
GO TO	GOTO	Reserved (Note 1)
IF		Reserved
INCLUDE		Reserved
INTERNAL	INT	Unreserved
ITERATE		
LEAVE		
LOOP		
KEYS		Unreserved
MACRO		Reserved
MARGINS	MAR	
MESSAGE	MSG	
MSGLEVEL		
NORESCAN		
NOSCAN		Reserved
NOTE		Reserved
OTHERWISE	OTHER	Reserved
PAGE		
PROCEDURE	PROC	Reserved
RESCAN		Reserved
RETURN		
RETURNS		Unreserved
REPEAT		
SELECT		Reserved
Reserved		
SKIP		
STATEMENT	STMT	Unreserved
THEN		Reserved
TO		
TRACE		
UNTIL		
WHEN		Reserved
WHILE		

Note 1: Neither GO nor TO when used by itself is reserved.

Data types and attributes

The data types supported by the macro facility are:

- Fixed point arithmetic data
- Character string data

For more information on data types refer to "Data types and attributes" on page 25.

Fixed point data

Fixed point data has the FIXED attribute and the implied base and precision of BINARY(31,0). Attributes other than FIXED cannot be specified. It is used for arithmetic computation.

Fixed point constants are written as FIXED DECIMAL constants, but without the decimal point. For more information on fixed point data, refer to "Coded arithmetic data and attributes" on page 30.

Character data

Character data has the CHARACTER attribute and the implied VARYING attribute with length that has no logical upper limit. Attributes other than CHARACTER cannot be specified. It is used for text manipulation.

Character constant and the X character constant are allowed. For more information on character data, refer to "Coded arithmetic data and attributes" on page 30.

Expressions

Expressions may use the concatenate operator and all arithmetic operators except exponentiation (**).

Conversions

Implicit conversions between FIXED and CHARACTER follow normal PL/I rules, except that FIXED always converts to exactly 8 characters. That is, for conversion purposes, FIXED data is treated as if it were declared as FIXED DECIMAL(5,0).

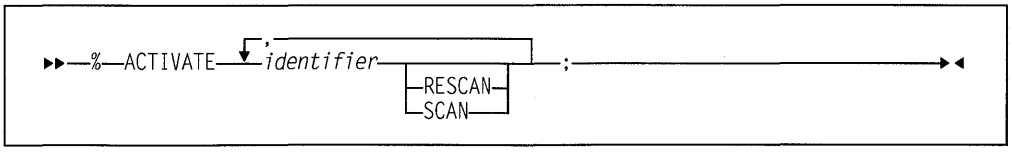
Macro facility statements

This section describes the macro facility statements in alphabetical order.

%ACTIVATE

The %ACTIVATE statement makes an identifier eligible for replacement by its value when the identifier is encountered in subsequent source text.

The syntax for the %ACTIVATE statement is:



Abbreviation: %ACT

identifier
specifies the name of a macro facility identifier.

RESCAN
specifies that the identifier is replaced as many times as necessary.

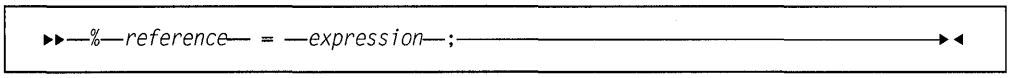
SCAN specifies that the identifier is replaced only once. (The keyword NORESCAN is a synonym for SCAN.)

Issuing the %ACTIVATE statement for an identifier that is already active has no effect, except to change the scanning status.

%assignment

A %assignment statement evaluates a macro facility expression and assigns the result to a macro facility variable. The variable need not be active nor have any specific scanning status.

The syntax is:

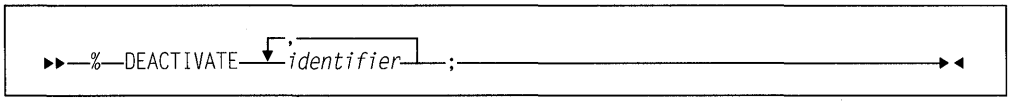


reference
specifies the name of a macro facility variable.

%DEACTIVATE

The %DEACTIVATE statement makes an identifier ineligible for replacement when the identifier is encountered in subsequent source text.

The syntax for the %DEACTIVATE statement is:



Abbreviation: %DEACT

identifier
specifies the name of the macro facility identifier.

The deactivation of an identifier causes it to be ineligible for replacement, but does not affect its value. Consequently, the reactivation of a deactivated identifier does not require the assignment of a replacement value.

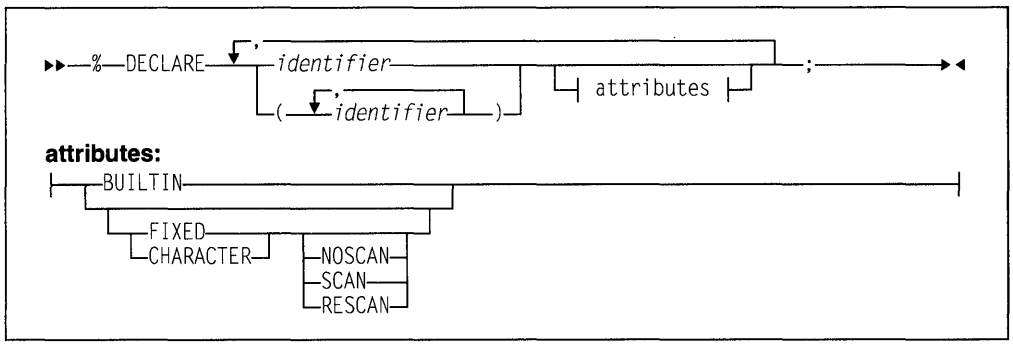
Deactivation of an inactive identifier has no effect.

%DECLARE

The %DECLARE statement establishes an identifier as a macro facility built-in function or variable, and specifies the identifier's scanning status.

Note: Before your application program refers to an identifier in macro facility statements, it must be declared, and the %DECLARE statement for the identifier must not be skipped as the result of a %GO TO statement, or an %IF-%THEN-%ELSE statement.

The syntax for the %DECLARE statement is:



Abbreviations: %DCL for %DECLARE, CHAR for CHARACTER

identifier
specifies the name of a macro facility identifier or built-in function

BUILTIN
specifies that the identifier is a macro facility built-in function. For more information on macro facility built-in functions refer to 458.

CHARACTER
specifies character data. For more information on macro facility character data, refer to 453.

FIXED specifies fixed point data. For more information on macro facility fixed point data, refer to "Fixed point data" on page 453.

RESCAN
specifies that the identifier is replaced as many times as necessary.

SCAN specifies that the identifier is replaced only once. (The keyword NORESCAN is a synonym for SCAN.)

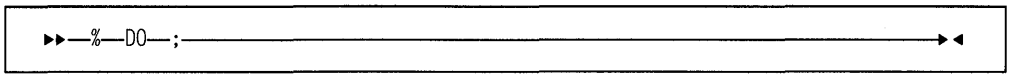
NOSCAN
specifies that the identifier is inactive and not replaced.

%DO

The %DO statement, and its corresponding %END statement, delimit a macro facility do group.

Macro facility statements and source text contained in the do group are processed only when the %DO statement is executed.

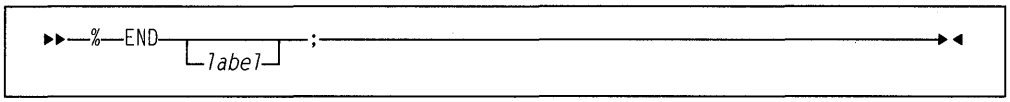
The syntax for the %DO statement is:



%END

The %END statement delimits a %DO statement.

The syntax for the %END statement is:

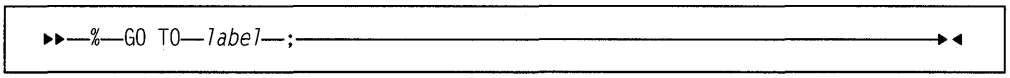


label must be the label of the most recent open %DO statement.

%GO TO

The %GO TO statement causes the scan to be resumed in the source text at the statement with the specified label. The target statement must appear after the %GO TO statement.

The syntax for the %GO TO statement is:



Abbreviation: %GOTO

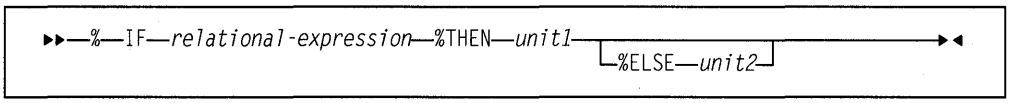
label label on the target statement

Macro facility statements between the %GO TO statement and the point where the scan is resumed are skipped and should be syntactically correct.

%IF

The %IF statement controls the flow of the macro facility scan.

The syntax for the %IF statement is:



relational-expression
specifies the relational expression that is evaluated to true or false. If true, the %THEN clause is executed and the %ELSE clause, if present, is skipped. If false, the %THEN clause is skipped and the %ELSE clause, if present, is executed.

%THEN clause
specifies unit1 as a %DO group or any macro facility statement, except %INCLUDE, %END, and %DECLARE. %INCLUDE and %DECLARE statements must be enclosed in a %DO group.

%ELSE clause
specifies unit2 as a %DO group or any macro facility statement other than %INCLUDE, %END, and %DECLARE. %INCLUDE and %DECLARE statements must be enclosed in a %DO group.

Scanning resumes immediately following the %IF statement, unless, of course, a %GO TO statement in one of the clauses causes the scan to resume elsewhere. Macro facility statements between the %IF statement and the point where the scan is resumed are skipped, and should be syntactically correct.

%IF statements can be nested in the same manner as described under "IF statement" on page 185 for nesting IF statements.

%INCLUDE

For the syntax and a description, refer to "%INCLUDE statement" on page 187.

The included file is treated normally, except that all macro facility statements, %DO groups, %GO TO targets, %THEN and %ELSE clauses must be fully contained with the same file.

The scan then continues with the first character in the included file. The included file is scanned in the same manner as the input source text. Hence, included files contribute to the output source text being formed.

%INCLUDE statements can be nested. In other words, included files can contain %INCLUDE statements.

Example

For example, assume that the external file named PAYRL contains the following:

```
declare 1 Payroll,
          2 Name,
            3 Last   character (30) varying,
            3 First  character (15) varying,
            3 Middle character (3) varying,
          2 Curr,
            3 (Regular, Overtime) fixed decimal (8,2),
          2 * char(0);
```

Then the following macro facility statements generate two structure declarations in the output source text. The only difference between them is their names, Cum_Pay and Payroll.

```
%declare Payroll character;
%Payroll='Cum_Pay';
%include PAYRL;
%deactivate Payroll;
%include PAYRL;
```

Execution of the first %INCLUDE statement incorporates the file in PAYRL into the input source text. When the macro facility scan encounters the identifier Payroll in this included file, it replaces it with the current value of the active macro facility

identifier Payroll, which is Cum_Pay. Further scanning of the included file results in no additional replacements. The macro facility scan then encounters the %DEACTIVATE statement and deactivates the macro facility identifier Payroll. When the second %INCLUDE statement is executed, the file in Payroll once again is incorporated into the input source text. This time, however, scanning of the included file results in no replacements.

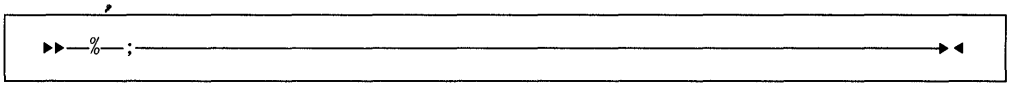
%NOTE

Refer to "%NOTE statement" on page 188.

%null

The %null statement causes no operation to be performed and does not modify the flow of macro facility scan.

The syntax for the %null statement is:



Macro facility built-in functions

These functions can be used in macro facility statements. The macro facility does not recognize these if they appear in the source text.

When the name of a macro facility built-in function is encountered, it is processed as a built-in unless the name has already been declared with attributes other than BUILTIN. Once implicitly or explicitly declared as a built-in function, all subsequent appearances of the name will be treated as built-in function references.

If the name is declared as something other than a built-in function, the name cannot be used as a built-in function.

Function	Description
COLLATE	Returns a character string consisting of all the 256 possible CHARACTER(1) values just once in an order known as the collating order.
COMMENT	Converts a character string into a valid PL/I comment.
COMPILETIME	Returns a character string containing the compilation date and time.
COUNTER	Returns a character string containing a numeric value that starts at '00001' and is incremented by 1 each time that COUNTER is referenced.
INDEX	Returns a value that indicates the starting position of a character string within another character string.
LENGTH	Returns the current length of a string.
MAX	Returns the maximum arithmetic value from a set of two or more arithmetic expressions.
MIN	Returns the minimum arithmetic value from a set of two or more arithmetic expressions.
QUOTE	Converts a character string to a quoted character string.

Function	Description
REPEAT	Repeats a character string a specified number of times.
SUBSTR	Returns a substring from a string.
SYSPARM	Returns the character string value of the SYSPARM compiler option.
SYSTEM	Returns a character string representation of the SYSTEM option value in effect.
SYSVERSION	Returns a character string indicating the version and release of the compiler.
TRANSLATE	Returns a character string that is the translation of an input character string.
VERIFY	Finds the position of the first character in the input string that is not present in the verification string.

COLLATE

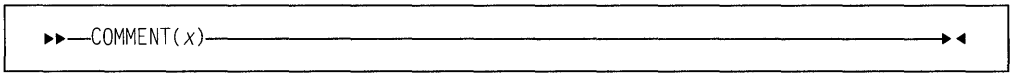
Refer to "COLLATE" on page 391.

COMMENT

COMMENT returns a character string that represents *x* as a valid PL/I comment.

If *x* contains /* or */ composite symbols, they are replaced by /> and </ respectively.

The syntax is:



x specifies the expression that is to be converted to a comment.

Example

In this example, the CHARACTER macro facility identifiers A and B are assigned character strings that are comments:

```
Test: proc;
   dcl S_Var1 character(35);
   dcl S_Var2 character(35);
  %dcl A character;
  %dcl B character;
  %A = comment('What');
  %B = comment('Name /* Ain''t it sweet? */');
   A;                                  /* Generates the next line      */
   /*What*/
   B                                   /* Generates the next line      */
   /*Name /> Ain't it sweet? </*/
 end;
```

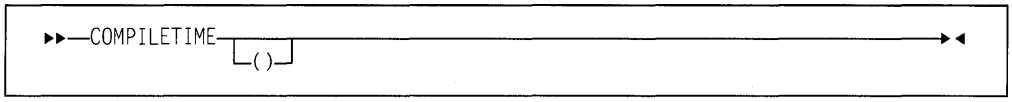
COMPILETIME

COMPILETIME returns a character string (of length 18) that contains the date and the time of compilation.

The format of the timestamp is DDbMMMbYYbHH.MI.SS, where:

b	is a blank character.
DD	is the current day. A leading zero in DD is replaced by a blank.
MMM	is the current month (abbreviated as JAN, FEB, and so on)
YY	is the current year.
HH	is the current hour.
.	is a separator.
MI	is the current minute.
SS	is the current second.

The syntax for COMPILETIME is:



Example

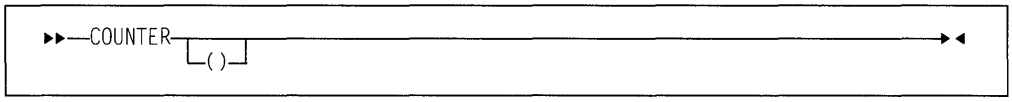
For the following example, if the compilation date and time are September 2, 1991, 2:15 am, the result is `2 SEP 91 02.15.00`.

```
Test: proc;
  %dcl C char;
  %C = compiletime;          /* = ' 2 SEP 91 02.15.00'        */
end;
```

COUNTER

COUNTER returns a character string (of length 5) containing a numeric value with leading zeros. The first reference returns a value of '00001'. Each subsequent reference returns a number one greater than the previous reference.

The syntax for COUNTER is:



Example

```
Test: proc;
  %dcl (C,d) char;
  %C = counter;          /* = '00001'           */
  %d = counter;          /* = '00002'           */
end;
```

INDEX

Refer to "INDEX" on page 407.

LENGTH

Refer to "LENGTH" on page 409.

MAX

Refer to "MAX" on page 412.

MIN

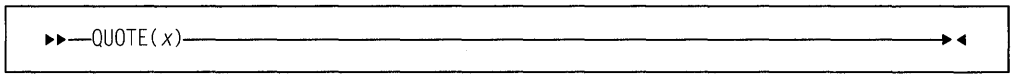
Refer to "MIN" on page 413.

QUOTE

QUOTE returns a character string that represents *x* as a valid quoted string.

If *x* contains single quotation marks, each is replaced by two consecutive single quotation marks.

The syntax is:



x character expression that is to be converted to a quoted string.

Example

```
This: proc;
  dcl Book char;
  %dcl Title char;
  %Title = "Shakespeare's ""Hamlet"""; /* Shakespeare's "Hamlet" */
  Book = quote(Title);                /* Generates ...          */
  Book = 'Shakespeare''s "Hamlet"';
end;
```

REPEAT

Refer to "REPEAT" on page 430.

SUBSTR

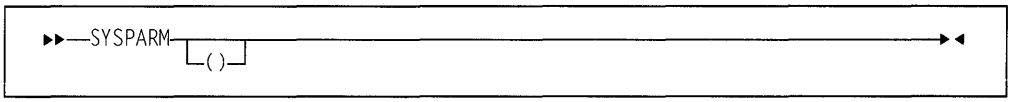
Refer to "SUBSTR" on page 439.

SYSPARM

SYSPARM returns the character string value of SYSPARM compiler option. See the description of the SYSPARM compiler option in the *PL/I Package/2 Programming Guide* for more information.

SYSPARM allows information external to the program to be accessed without modifying the source program.

The syntax is:



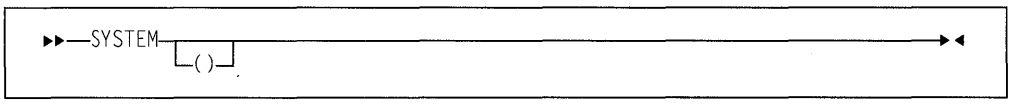
Example

```
    pli mypgm (sysparm('TEST') macro

Mypgm: proc;
  dcl Dept_num char(11);
  %dcl Group_id char;
  %dcl Group_num char;
  %if (sysparm = 'DEVELOPMENT')   %then          /* false */
   %Group_id = 'G32/001/DEV';
  %else
    %do;
      %if sysparm = 'SHIPPING'  %then           /* false */
       %Group_id = 'G35/002/SHP';
      %else
        %do;
          %if (sysparm = 'TEST')  %then         /* true  */
           %Group_ID = 'G37/006/TST';
          %else
           %Group_id = 'G99/000/MSC';
        %end;
    %end;
  Group_num = quote(Group_id_);
  Dept_num = Group_num;        /* Generates ...       */
  Dept_num = 'G37/006/TST';
end;
```

SYSTEM

SYSTEM returns a character string that contains of the value of the SYSTEM compiler option that is in effect. See the description of the SYSTEM compiler option in the *PL/I Package/2 Programming Guide*.

The syntax is:



Example

For the following example, assume that OS/2 is active.

```
%process system(os2);
This: proc;
  %dcl P_Var1 char;
  dcl S_Var1 character(20);
  %P_Var1 = Quote(system);
  S_Var1 = P_Var1;                /* Generates ...       */
  S_VAR1 = ' OS2 ';
end;
```

SYSVERSION

SYSVERSION returns a character string (of length 31) indicating the name of the compiler and the version, release, and modification level.

The format of the string is:

```
nnnnnnnnbvvrrmmbxxxxxxxxxxxxxxx
```

in which:

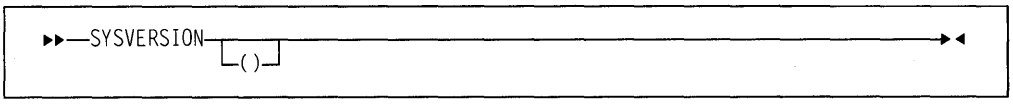
nnnnnnnn
is the compiler name (PL/I)

b indicates a single blank

vvrrmm
is the compiler version (*vv*), release (*rr*), and modification level (*mm*)

xx...xx Reserved for future use

The syntax is:



Example

```
This: proc;
  %dcl P_Var1 char;
  dcl S_Var1 character(31);
  %P_Var1 = Quote(sysversion);

  S_Var1 = P_Var1;  /* Generates ...       */
  S_Var1 = 'PL/I     V1R1M0 xxxxxxxxxxxxxxx';
end;
```

TRANSLATE

Refer to "TRANSLATE" on page 442.

VERIFY

Refer to "VERIFY" on page 447.

Macro facility examples

Example 1

Assume the input source text contains the following statements.

```
%declare A character, B fixed;
%A = 'B+C';
%B = 2;
X = A;
```

The macro facility produces the following output source text.

```
X =      2+C;
```

The macro facility statements declare A and B with the default status of RESCAN, assign the character string 'B+C' to A, and assign the constant 2 to B.

The fourth line is source text. The current value of A, which is 'B+C', replaces A in the output source text. But this string contains the macro facility identifier B. Upon rescanning B, the macro facility finds that it is active. Hence, the value 2 replaces B in the output source text. Since identifier B has a data type of FIXED, it converted to an 8-character character string ('bbbbbbb2').

Further rescanning shows that 2 cannot be replaced. Scanning resumes with +C which, again, cannot be replaced.

If, in the above example, the macro facility identifier A was activated by the following statement:

```
%activate A norescan;
```

The output source text would be:

```
X = B+C;
```

Example 2

The input source text contains the following statements.

```
%declare I fixed, T character;
%deactivate I;
%I = 15;
%T = 'A(I)';
S  = I*T*3;
%I = I+5;
%activate I;
%deactivate T;
R  = I*T*2;
```

The output source text is as follows. (Replacement blanks for conversion from fixed to character are not shown.)

```
S = I*A(I)*3;
R = 20*T*2;
```

Appendix A. Limits

Figure 71 summarizes the implementation limits for the PL/I Package/2 language elements.

Figure 71 (Page 1 of 3). Language element limits

Language Element	Description	Limit
Arrays	Maximum number of dimensions for an array	15
	Minimum lower bound	-2147483647
	Maximum upper bound	+2147483646
Structures	Maximum number of levels in a structure	15
	Maximum level number in a structure	255
Arithmetic Precisions	Maximum precision for FIXED DECIMAL	31
	Maximum precision for FIXED BINARY	31
	Maximum precision for FLOAT DECIMAL	18
	Maximum precision for FLOAT BINARY	64
	Maximum scale factor for FIXED data	127
	Minimum scale factor for FIXED data	-128
String and AREA Variables	Maximum length of CHARACTER	32767
	Maximum length of BIT	32767
	Maximum length of GRAPHIC	16383
	Maximum size of AREA	16777216
Built-In Functions	Maximum number of arguments to the MAX and MIN functions	64
	Maximum values for the precision (p) in the ADD, BINARY, DECIMAL, DIVIDE, FIXED, FLOAT, MULTIPLY, PRECISION, and SUBTRACT functions	same as corresponding limit for arithmetic precision
	Maximum values for the scale (q) in the ADD, BINARY, DECIMAL, DIVIDE, FIXED, MULTIPLY, PRECISION, and SUBTRACT functions	same as corresponding limit for arithmetic precisions
	Maximum number of digits (N) in the CEIL, FLOOR, MAX, MIN, MOD, ROUND and TRUNC functions	same as corresponding limit for arithmetic precisions

Figure 71 (Page 2 of 3). Language element limits

Language Element	Description	Limit
Program Size	Maximum length of an identifier	31
	Maximum number of procedures in a program	255
	Maximum number of lexical units (keywords, identifiers, delimiters, etc) before a statement type can be resolved	511
	Maximum number of DEFAULT-statements in a block	31
	Maximum number of %PUSH statements	63
	Maximum number of LIKE-attributes in a block	63
	Maximum number of output expressions in a data-list	60
	Maximum number of repetitive DO-specifications in a data-list	15
	Maximum size of a data aggregate	2147483647
	Maximum number of arguments in a CALL or function reference	255
	Maximum number of parameters for a procedure	255
	Maximum nesting of factored attributes	15
	Maximum nesting of BEGIN and PROCEDURE statements	30
	Maximum nesting of DO-groups	49
	Maximum nesting of IF statements	49
	Maximum nesting of SELECT-statements	49

Figure 71 (Page 3 of 3). Language element limits

Language Element	Description	Limit
Miscellaneous	Maximum number of picture characters in a character picture	511
	Maximum number of bytes in a numeric picture	253
	Maximum number of numeric picture characters in a numeric picture	15
	Maximum number of bytes in the external representation of CHARACTER, X, BIT, BX, GRAPHIC, GX and M string constants The external representation includes all quotes and string suffixes. For example, the string '01010110'B has 11 bytes in its external specification, but only 1 byte in its internal representation. Similarly, the string 'Ain''t Misbehavin'' has 21 bytes in its external specification, but only 17 in its internal representation.	3072
	Maximum length for a KEYTO character string	120
	Maximum length for a KEYTO graphic string	60
	Maximum line size for LINESIZE	32,000
	Minimum line size for LINESIZE	1
	Maximum page size for PAGESIZE	32,767
	Minimum page size for PAGESIZE compiler option	1
	Maximum size of DISPLAY character string	126
	Maximum DISPLAY reply message.	72 bytes
	Range of IEEE normalized floating-point numbers	+3.30E-4932 to +1.21E+4932, 0, -3.30E-4932 to -1.21E+4932
	Range of hex floating-point numbers	+10E-78 to +10E75, 0, -10E-78 to +10E+75

Bibliography

IBM SAA AD/Cycle PL/I Package/2 publications

- *PL/I Package/2 Fact Sheet*, GC26-3090
- *PL/I Package/2 Programming Guide*, SC26-4822
- *PL/I Package/2 Language Reference*, SC26-4823
- *PL/I Package/2 Reference Summary*, SX26-3793
- *PL/I Package/2 Installation*, SX26-3822
- *PL/I Package/2 Language Environment Run-Time Messages*, SC26-3133
- *PL/I Package/2 Licensed Program Specifications*, GC26-4821

IBM OS PL/I Version 2 publications

- *Programming Guide*, SC26-4307
- *Programming: Language Reference*, SC26-4308
- *Programming: Reference Summary*, SX26-3759

IBM Systems Application Architecture publications

- *An Overview*, GC26-4341
- *Common User Access: Panel Design and User Interaction*, SC26-4351
- *Writing Applications: A Design Guide*, SC26-4362
- *CPI PL/I Reference*, SC26-4381
- *CPI Communications Reference*, SC26-4399
- *CPI Database Reference*, SC26-4353
- *CPI Dialog Reference*, SC26-4356
- *CPI Presentation Reference*, SC26-4359
- *CPI Procedures Language Reference*, SC26-4358
- *CPI Query Reference*, SC26-4349

IBM OS/2 2.0 technical library

- *Application Design Guide*, S10G-6260
- *Programming Guide, Volume 1*, S10G-6261
- *Programming Guide, Volume 2*, S10G-6494
- *Programming Guide, Volume 3*, S10G-6495
- *Information Presentation Facility Guide and Reference*, S10G-6262
- *System Object Model Guide and Reference*, S10G-6309
- *Control Program Programming Reference*, S10G-6263
- *Presentation Manager Programming Reference Volume 1*, S10G-6264
- *Presentation Manager Programming Reference Volume 2*, S10G-6265
- *Presentation Manager Programming Reference Volume 3*, S10G-6272
- *Physical Device Driver Reference*, S10G-6266
- *Virtual Device Driver Reference*, S10G-6310
- *Presentation Manager Driver Reference*, S10G-6267
- *Procedures Language 2/REXX Reference*, S10G-6268
- *Procedures Language 2/REXX User's Guide*, S10G-6269
- *SAA Common User Access Guide to User Interface Design*, SC34-4289
- *SAA Common User Access Advanced User Interface Design Guide*, SC34-4290

Other books you might need

The following list contains the titles of IBM books that you might find helpful.

- *IBM BookManager READ/2: General Information*, GB35-0800
- *IBM BookManager READ/2: Getting Started and Quick Reference*, SX76-0146
- *IBM BookManager READ/2: Displaying Online Books*, SB35-0801
- *IBM BookManager READ/2: Installation*, GX76-0147

The following PL/I Package/2 publications can be ordered as a set using the bill of forms order number SBOF–3014: *Programming Guide*, *Reference Summary*, *Language Reference*, and *Language Environment Run-Time Messages*.

You can order these publications through your IBM representative.

Glossary

A

access. The act of referencing or retrieving data.

action specification. In an ON statement, the on-unit or the single keyword SYSTEM, either of which specifies the action to be taken whenever an interrupt results from the raising of the named condition. The action specification can also include the keyword SNAP.

activate (a block). To initiate the execution of a block. A procedure block is activated when it is invoked at an entry point. A begin block is activated when it is encountered in the normal flow of control, including a branch.

active. The establishment of the validity for replacement of the value of a variable or the returned value of an entry name. The first activation must be the result of the appearance of the name in a %DECLARE statement. If an active variable or entry name is made inactive by the %DEACTIVATE statement it may be activated again by an %ACTIVATE statement.

active. (1) The state of a block after activation and before termination. (2) The state in which a preprocessor variable or preprocessor entry name is said to be when its value can replace the corresponding identifier in source program text.

additive attribute. A file description attribute for which there are no defaults, and which, if required, must be stated explicitly or implied by another explicitly stated attribute. Contrast with *alternative attribute*.

adjustable extent. Bound (of an array), length (of a string), or size (of an area) that might be different for different generations of the associated variable. Adjustable extents are specified as expressions or asterisks (or by REFER options for based variables), which are evaluated separately for each generation. They cannot be used for static variables.

aggregate. See *data aggregate*.

aggregate expression. An array, structure, or union expression.

aggregate type. For any item of data, the specification whether it is structure, union, or array.

allocated variable. A variable with which internal storage is associated and not freed.

allocation. (1) The reservation of main storage for a variable. (2) A generation of an allocated variable.

alignment. The storing of data items in relation to certain machine-dependent boundaries (for example, a fullword or halfword boundary).

alphabetic character. Any of the characters A through Z of the English alphabet and the alphabetic extenders #, $, and @ (which may have a different graphic representation in different countries).

alphameric character. An alphabetic character or a digit.

alternative attribute. A file description attribute that is chosen from a group of attributes. If none is specified, a default is assumed. Contrast with *additive attribute*.

ambiguous reference. A reference that is not sufficiently qualified to identify one and only one name known at the point of reference.

area. A declared portion of storage identified by an area variable with the AREA attribute. Its values may only be areas.

area variable. A variable with the AREA attribute. Its values may only be areas.

argument. An expression in an argument list as part of an invocation of a subroutine or function.

argument list. A parenthesized list of one or more arguments, separated by commas, following an entry name constant, an entry name variable, a generic name, or a built-in function name. The list is passed to the parameters of the entry point.

arithmetic comparison. A comparison of signed numeric values. See also *bit comparison, character comparison*.

arithmetic constant. A fixed-point constant or a floating-point constant. Although most arithmetic constants can be signed, the sign is not part of the constant.

arithmetic conversion. The transformation of a value from one arithmetic representation to another.

arithmetic data. Data that has the characteristics of base, scale, mode, and precision. Coded arithmetic data and pictured numeric character data are included.

arithmetic operators. Either of the prefix operators + and –, or any of the following infix operators: + – * / **

arithmetic picture data. Decimal picture data containing the following types of picture specification characters:

- Decimal digit characters
- Zero suppression characters
- Sign and currency symbol characters
- Insertion characters
- Commercial characters
- Exponent characters.

array. A named, ordered, collection of one or more data elements with identical attributes, grouped into one or more dimensions.

array expression. An expression whose evaluation yields an array of values.

array of structures. An ordered collection of identical structures specified by giving the dimension attribute to a structure name.

array variable. A variable that represents an aggregate of data items that must have identical attributes. Contrast with *structure variable*.

ASCII. (American National Standard Code for Information Interchange). The standard code, using a coded character set consisting of 7-bit coded characters (the 8th bit for parity check) that is used for information interchange among data processing systems, data communication systems, and associated equipment. The ASCII set consists of control characters and graphic characters.

assignment. The process of giving a value to a variable.

asynchronous operation. The overlap of an input/output operation with the execution of statements or the concurrent execution of procedures using multiple flows of control for different tasks.

attachment of a task. The invocation of a procedure and the establishment of a separate flow of control to execute the invoked procedure (and procedures it invokes) asynchronously, with execution of the invoking procedure.

attention. An occurrence, external to a task, that could cause an interrupt to the task.

attribute. (1) A descriptive property associated with a name to describe a characteristic represented. (2) A descriptive property used to describe a characteristic of the result of evaluation of an expression.

automatic storage allocation. The allocation of storage for automatic variables.

automatic variable. A variable whose storage is allocated automatically at the activation of a block and released automatically at the termination of that block.

B

base. The number system in which an arithmetic value is represented.

base element. The name of a structure member that is not a minor structure.

base item. The automatic, controlled, or static variable or the parameter upon which a defined variable is defined. The name may be qualified and/or subscripted.

based reference. A reference that has an explicit or implicit locater qualifier as its object. The name contained in a based reference necessarily refers to a based variable (that is, variable declared with the BASED attribute).

based storage allocation. The allocation of storage for based variables.

based variable. A variable that provides attributes for data (for example, data located in a buffer) for which the storage address is provided by a locater. Multiple generations of the same variable are accessible. It does not identify a fixed location in storage.

begin-block. A collection of statements delimited by BEGIN and END statements. It is part of a program that delimits the scope of names. A begin-block is activated either by error-handling ON-conditions or through the normal flow of control including any branch resulting from a GOTO statement.

binary. A number system whose only numerals are 0 and 1.

binary fixed-point value. An integer consisting of binary digits and having an optional binary point. Contrast with *decimal fixed-point value*.

binary floating-point value. An approximation of a real number in the form of a significand, which can be considered as a binary fraction, and an exponent, which can be considered as an integer exponent to the base of 2. Contrast with *decimal floating-point value*.

bit. (1) A string composed of zero or more bits. (2) The smallest amount of space of computer storage.

bit comparison. A left-to-right, bit-by-bit comparison of binary digits. See also *arithmetic comparison, character comparison*.

bit constant. A series of binary digits enclosed in apostrophes and followed immediately by B or B1. Contrast with *character constant*.

bit string constant. A series of hexadecimal digits enclosed in apostrophes and followed immediately by B4 or BX.

bit value. A sequence of binary digits stored in consecutive bits.

bit string. A string composed of zero or more bits.

bit string operators. The logical operators ¬ (not), & (and), and | (or).

block. A sequence of statements, processed as a unit, that specifies the scope of names and the allocation of storage for names declared within it.

block heading statement. The PROCEDURE, BEGIN or PACKAGE statement that heads a block of statements.

bound-pair list. A list specifying the lower and upper bounds of the valid indexes used to select elements of an array.

bounds. The upper and lower limits of an array dimension.

break character. The underscore symbol (_). It can be used to improve the readability of identifiers. For instance, a variable could be called OLD_INVENTORY_TOTAL instead of OLDINVENTORYTOTAL.

built-in function. A predefined function supplied by the language, such as a commonly used arithmetic function or a function needed by language facilities (for example, a function for manipulating strings or converting data). It is called by a built-in function reference, such as SQRT (square root).

built-in function reference. A built-in function name, having an optional and possibly empty argument list, that represents the value returned by the built-in function.

buffer. Intermediate storage, used in input/output operations, into which a record is read during input and from which a record is written during output.

bug. Generic term that encompasses anything that a program does that it was not designed to do.

C

call. (1) (verb) To invoke a subroutine use the CALL statement or CALL option (2) (noun) The using of the CALL statement or CALL option to invoke a subroutine.

character comparison. A left-to-right, character-by-character comparison according to the collating sequence. See also *arithmetic comparison, bit comparison*.

character constant. A sequence of characters enclosed in apostrophes; for example, 'CONSTANT'.

character set. A defined collection of characters. See *language character set* and *data character set*.

character string. A string composed of zero or more characters enclosed in single quotes.

character string picture data. Data described by a picture specification that must have at least one A or X picture specification character.

closing (of a file). The dissociation of a file from a data set.

coded arithmetic data. Data items that represent numeric values and are characterized by their base (decimal or binary), scale (fixed-point or floating-point), and precision (the number of digits each can have). This data is stored in a form that is acceptable, without conversion, for arithmetic calculations.

code inspection. Debugging technique that involves selecting a written piece of source code and reading through it from the perspective of the computer.

combined nesting depth. The sum of all PROCEDURE/BEGIN/ON, DO, SELECT, and IF...THEN...ELSE nestings in the program.

comment. A string of zero or more characters used for documentation that are delimited by /* and */.

commercial character.

- CR (credit) picture specification character
- DB (debit) picture specification character

comparison operator. An operator that can be used in an arithmetic, string, or logical relation to indicate the comparison to be done between the terms in the relation. The comparison operators are:

= (equal to)
> (greater than)
< (less than)
>= (greater than or equal to)
<= (less than or equal to)
¬= (not equal to)
¬> (not greater than)
¬< (not less than).

compile time. In general, the time during which a source program is translated into an object module. In PL/I, it is the time during which a source program can be altered (preprocessed), if desired, and then translated into an object program.

compiler options. Keywords that are specified to control certain aspects of a compilation, such as: the nature of the load module generated by the compiler, the types of printed output to be produced, the efficient use of the compiler, and the destination of error messages.

complex data. Arithmetic data, each item of which consists of a real part and an imaginary part.

composite operator. An operator that consists of more than one special character, such as <=, **, and /*.

compound statement. A statement that contains other statements. In PL/I, IF, SELECT, and ON are the only compound statements. See *statement body*.

concatenation. The operation that joins two strings in the order specified, forming one string whose length is equal to the sum of the lengths of the two original strings. It is specified by the operator ||.

condition. An exceptional situation, either an error (such as an overflow), or an expected situation (such as the end of an input file). When a condition is raised (detected), the action established for it is processed. See also *established action* and *implicit action*.

condition list. A list of one or more condition prefixes.

condition name. Name of a PL/I-defined or programmer-defined condition

condition prefix. A parenthesized list of one or more condition names prefixed to a statement. It specifies whether the named conditions are to be enabled. A condition list may be attached to any statement except DECLARE, DEFAULT, ENTRY, or % statements.

connected aggregate. An array or structure that has no inherited dimensions.

connected reference. A reference to connected storage. It must be apparent, prior to execution of the program, that the storage is connected.

connected storage. Main storage of an uninterrupted linear sequence of items that can be referred to by a single name.

constant. (1) An arithmetic or string data item that does not have a name and whose value cannot change. (2) An unsubscripted label prefix of a file name or an entry name.

constant reference. A value reference which has a constant as its object

contained text. (1) All text in a procedure (including nested procedures) except its entry names and condition prefixes of the PROCEDURE statement; (2) All text in a begin block except labels and condition prefixes of the BEGIN statement that heads the block. Internal blocks are contained in the external procedure.

contextual declaration. The appearance of an identifier that has not been explicitly declared (in a DECLARE statement, as a label prefix, or in a parameter list), in a context that allows the association of specific attributes with the identifier.

control character. A nonprinting character in a character set whose occurrence in a particular context specifies a control function.

control code. A code point and its assigned control function meaning, for example, "end of transmission." For 7-bit codes such as ASCII, the first 32 code points are reserved for control purposes. For EBCDIC 8-bit codes, the first 64 code points are reserved.

control format item. A specification used in edit-directed transmission to specify positioning of a data item within the stream or printed page.

control variable. A variable that is used to control the iterative execution of a group, as in a DO statement.

controlled parameter. A parameter for which the CONTROLLED attribute is specified in a declare statement. It can be associated only with arguments that have the CONTROLLED attribute.

controlled storage allocation. The allocation of storage for controlled variables.

controlled variable. A variable whose allocation and release are controlled by the ALLOCATE and FREE statements, with access to the current generation only.

conversion. The transformation of a value from one representation to another to conform to a given set of attributes.

cross section of an array. The elements represented by the extent of at least one dimension of an array. An asterisk in the place of a subscript in an array reference indicates the entire extent of that dimension.

current generation. That generation (of an automatic or controlled variable) currently available by reference to the name of the variable.

D

data testing. Debugging technique that involves using test data to verify that a program operates as designed.

DBCS. Double-byte character set. Each hexadecimal, 2-byte DBCS code identifies a double-byte character. For example, Japanese extended Kanji characters are encoded in DBCS.

data. Representation of information or of value in a form suitable for processing.

data aggregate. A group of data items that can be referred to either individually or collectively. The types of aggregates are: arrays, unions and structures.

data attribute. A keyword that specifies the type or kind of data to process.

data character set. All of those characters whose representation is recognized by the computer in use.

data-directed transmission. The type of stream-oriented transmission in which data is transmitted as a group, ended by a semicolon, where each item is of the form:

```
name = constant
```

data item. A single unit of data. It is synonymous with *element*.

data list. In stream oriented data transmission, a parenthesized list of the data items used in GET EDIT and PUT EDIT statements. Contrast with *format list*.

data set. A collection of data external to the program that can be accessed by reference to a single file name.

data specification. The portion of a stream-oriented data transmission statement that specifies the mode of transmission (DATA, LIST, or EDIT) and includes the data list(s) and, for edit-directed mode, the format list(s).

data stream. Data being transferred from or to a data set by stream-oriented transmission, as a continuous stream of data elements in character form.

data transmission. The transfer of data from a data set to the program or vice versa.

data type. A set of data attributes.

deactivated. The state in which an identifier is said to be when its value cannot replace the corresponding identifier in source program text.

debugging. Process of removing bugs from a program.

decimal. The number system whose numerals are 0 through 9.

decimal digit character. The picture specification character 9.

decimal fixed-point constant. A constant consisting of one or more decimal digits with an optional decimal point.

decimal fixed-point value. A rational number consisting of a sequence of decimal digits with an assumed position of the decimal point. Contrast with *binary fixed-point value*.

decimal floating-point constant. A value made up of a significand that consists of a decimal fixed-point constant, and an exponent that consists of the letter E followed by an optionally signed integer constant not exceeding three digits.

decimal floating-point value. An approximation of a real number, in the form of a significand, which can be considered as a decimal fraction, and an exponent, which can be considered as an integer exponent to the base of 10. Contrast with *binary floating-point value*.

decimal picture data. Arithmetic picture data specified by picture specification characters containing the following types of picture specification characters:

- Decimal digit characters
- The virtual point picture character
- Zero-suppression characters
- Sign and currency symbol characters
- Insertion characters
- Commercial characters
- Exponent characters.

declaration. (1) The establishment of an identifier as a name and the specification of a set of attributes (partial or complete) for it. (2) A source of attributes of a particular name.

default. Describes a value, attribute, or option that is assumed when none has been specified.

defined item. A variable which is to be associated with some or all of the storage associated with the designated base variable.

delimit. To enclose one or more items or statements with preceding and following characters or keywords.

delimiter. All comments and the following characters: percent, parentheses, comma, period, semicolon, colon, assignment symbol and blank. They define the limits of identifiers, constants, picture specifications, and keywords.

descriptor. A skeletal form of a declaration in which declared names are omitted. See *parameter descriptor.*

digit. One of the characters 0 through 9.

dimension attribute. An attribute that specifies the number of dimensions of an array and indicates the bounds of each dimension.

directive. A statement that directs the operation of the compiler.

disabled. The state of a condition in which no interrupt occurs and no established action commences.

do group. A sequence of statements delimited by a DO statement and ended by its corresponding END statement, used for control purposes. Contrast with *block.*

do-loop. See *iterative do-group.*

dummy argument. Temporary storage that is created automatically to hold the value of an argument that cannot be passed by reference.

dump. Printout of all or part of the storage used by a program as well as other program information, such as a trace of an error's origin.

E

EBCDIC. (Extended Binary-Coded Decimal Interchange Code). A coded character set consisting of 8-bit coded characters.

edit-directed transmission. The type of stream-oriented transmission in which data appears as a continuous stream of characters and for which a format list is required to specify the editing desired for the associated data list.

element. A single item of data as opposed to a collection of data items such as an array; a scalar item.

element expression. An expression whose evaluation yields an element value.

element variable. A variable that represents an element; a scalar variable.

elementary name. See *base element.*

enabled. The state of a condition in which a particular on-unit results in a program interrupt that causes an established action to commence.

entry constant. The label prefix of a PROCEDURE statement (an entry name).

entry data item. A data item that represents an entry point to a procedure.

entry expression. An expression whose evaluation yields an entry name.

entry name. (1) An identifier that is explicitly or contextually declared to have the ENTRY attribute (unless the VARIABLE attribute is given) or (2) Has the value of an entry variable with the ENTRY attribute implied.

entry point. A point in a procedure at which it may be invoked.

entry reference. An entry constant, an entry variable reference, or a function reference that returns an entry value.

entry variable. A variable to which an entry value can be assigned. It must have both the ENTRY and VARIABLE attributes.

entry value. The entry point represented by an entry constant; the value includes the environment of the activation that is associated with the entry constant.

environment (of an activation). Information associated with and used in the invoked block regarding data declared outside the block.

environment (of a label constant). Identity of the particular activation of a block to which a reference to a statement-label constant applies. This information is determined at the time a statement-label constant is passed as an argument or is assigned to a statement-label variable, and it is passed or assigned along with the constant.

established action. The action taken when a condition is raised. See also *implicit action* and *ON-statement action.*

epilogue. Those processes that occur automatically at the termination of a block or task.

evaluation. The reduction of an expression to a single value, an array of values, or a structured set of values.

explicit declaration. The appearance of an identifier (a name) in a DECLARE statement, as a label prefix, or in a parameter list. Contrast with *implicit declaration*.

exponent characters. The following picture specification characters:

1. K and E, which are used in floating-point picture specifications to indicate the beginning of the exponent field.
2. F, the scaling factor character, specified with an integer constant that indicates the number of decimal positions the decimal point is to be moved from its assumed position to the right (if the constant is positive) or to the left (if the constant is negative).

expression. (1) A notation, within a program, that represents a value, an array of values, or a structured set of values; (2) A constant or a reference appearing alone, or a combination of constants and/or references with operators.

extended alphabet. Upper and lower case alphabetic characters A through Z, $, @ and #.

extent. (1) The range indicated by the bounds of an array dimension, by the length of a string, or by the size of an area (2) The significant allocations in an area.

external name. A name (with the EXTERNAL attribute) whose scope is not necessarily confined only to one block and its contained blocks.

external procedure. A procedure that is not contained in any other procedure.

extralingual character. Characters (such as $, @, and #) that are not classified as alphanumeric or special. This group can include characters that are determined with the NAME compiler option.

F

factoring. The application of one or more attributes to a parenthesized list of names in a DECLARE statement, eliminating the repetition of identical attributes for multiple names

field (in the data stream). That portion of the data stream whose width, in number of characters, is defined by a single data or spacing format item.

field (of a picture specification). Any character-string picture specification or that portion (or all) of a numeric character picture specification that describes a fixed-point number.

file. A named representation, within a program, of a data set or data sets. A file is associated with the data set(s) for each opening.

file constant. A name declared for a file and for which a complete set of file description attributes exists during the time that the file is open, and with which each file must be associated.

file description attributes. Keywords that describe the individual characteristics of each file constant. See also *alternative attribute* and *additive attribute*.

file expression. An expression whose evaluation yields a file name.

file name. A name declared for a file.

file variable. A variable to which file constants can be assigned. It has the attributes FILE and VARIABLE and cannot have any of the file description attributes.

fixed-point constant. See *arithmetic constant*.

floating-point constant. See *arithmetic constant*.

flow of control. Sequence of execution.

footprints. Output markers in a program that indicate where a program is in its execution flow or display the values of identifiers. Used as a debugging technique.

format item. A specification used in edit-directed data transmission to describe the representation of a data item in the stream (data format item) or the specific positioning of a data item within the stream (control format item).

format list. In stream oriented data transmission, a list specifying the format of the data item on the external medium. Contrast with *data list*.

fully-qualified name. A qualified name that includes all the names in the hierarchical sequence above the structure member to which the name refers, as well as the name of the member itself.

function (programmer-specified or built-in). A procedure that has a RETURNS option in the PROCEDURE statement. It is invoked by the appearance of one of its entry names in a function reference and it returns a scalar value to the point of reference. Contrast with *subroutine*.

function reference. An entry constant or an entry variable, either of which must represent a function, followed by a possibly empty argument list. Contrast with *subroutine call*.

G

generation (of a variable). The allocation of a static variable, a particular allocation of a controlled or automatic variable, or the storage indicated by a particular locator qualification of a based variable or by a defined variable or parameter.

generic descriptor. A descriptor used in a GENERIC attribute.

generic key. A character string that identifies a class of keys. All keys that begin with the string are members of that class. For example, the recorded keys "ABCD," "ABCE," and "ABDF," are all members of the classes identified by the generic keys "A" and "AB," and the first two are also members of the class "ABC"; and the three recorded keys can be considered to be unique members of the classes "ABCD," "ABCE," "ABDF," respectively.

generic name. The name of a family of entry names. A reference to the generic name is replaced by the entry name whose parameter descriptors match the attributes of the arguments in the argument list at the point of invocation.

group. A collection of statements contained within larger program units. A group is either a do-group or a select-group and it can be used wherever a single statement can appear, except as an on-unit.

H

hexadecimal. Pertaining to a numbering system with a base of sixteen; valid numbers use the digits 0 through 9 and the characters A through F, where A represents 10 and F represents 15.

I

identifier. A string of characters, not contained in a comment or constant, and preceded and followed by a delimiter. The first character of the identifier must be one of the 29 extended alphabetic characters. The others, if any, can be extended alphabetic, digit, or the break character.

IEEE. Institute of Electrical and Electronics Engineers.

implicit. The action taken in the absence of an explicit statement.

implicit action. The action established for a condition when the program is activated and that remains established unless overridden by disabling the condition or by the processing of an ON statement for the same condition. Contrast with *ON-statement action*.

implicit declaration. The appearance of a name in a program when it has not been explicitly declared. A default set of attributes is assumed for the name.

implicit opening. The opening of a file as the result of an input or output statement other than the OPEN statement.

infix operator. An operator that appears between two operands.

inherited dimensions. For a structure field, those dimensions that are derived from the containing structures. If the structure field is a scalar variable, the dimensions consist entirely of its inherited dimensions. If the structure field is an array, its dimensions consist of its inherited dimensions plus its explicitly declared dimensions. A structure field with one or more inherited dimensions is referred to as an unconnected aggregate. Contrast with *connected aggregate*.

initial procedure. See *main procedure*.

input/output. The transfer of data between auxiliary medium and main storage.

insertion point character. A picture specification character that is, on assignment of the associated data to a character string, inserted in the indicated position. When used in a P-format item for input, an insertion character serves as a checking picture character.

instruction pointer. A pointer that provides addressability for a machine instruction in a program.

integer. A sequence of digits.

integral boundary. The multiple of any 8-bit unit of information on which data can be aligned.

interleaved array. An array whose name refers to non-connected storage.

interleaved subscripts. A subscript notation used with subscripted qualified names in which not all of the necessary subscripts immediately follow the same component name.

internal block. A block that is contained in another block.

internal name. A name that is not known outside the block in which it is declared.

internal procedure. A procedure that is contained in another block. Contrast with *external procedure*.

internal text. Text that is contained in a block, but not contained in any other block nested within it.

interrupt. The redirection of the program's flow of control (possibly temporary) as the result of raising a condition or attention.

invocation. The activation of a procedure.

invoke. To activate a procedure.

invoked procedure. A procedure that has been activated.

invoking block. A block containing a statement that activates a procedure.

iteration factor. (1) In an INITIAL attribute specification, an expression that specifies the number of consecutive elements of an array that are to be initialized with a given constant. (2) In a format list, an expression that specifies the number of times a given format item or list of items is to be used in succession.

iterative do-group. A do-group whose DO statement specifies a control variable and/or a WHILE or UNTIL option.

K

key. Data that identifies a record within a direct-access data set. See *source key* and *recorded key*.

keyword. An identifier that has a specific meaning to the compiler when used in a defined context.

keyword statement. A simple statement that begins with a keyword, indicating the function of the statement.

known. (applied to a name) Recognized with its declared meaning. A name is known throughout its scope.

L

label. A name used to identify a statement other than a PROCEDURE or an ENTRY statement. A statement label.

label data item. A label constant or the value of a label variable.

label constant. A name written as the label prefix of a statement (other than PROCEDURE or ENTRY) so that, during execution, program control can be transferred to that statement through a reference to its label prefix.

label list (of a statement). All of the label prefixes of a statement.

label list (of a label variable declaration). A parenthesized list of one or more statement-label constants immediately following the keyword LABEL to specify the range of values that the declared variable may have. Names in the list are separated by commas. When specified for a label array, it indicates that each element of the array may assume any of the values listed but no other.

label prefix. A label prefixed to a statement.

label variable. A variable declared with the LABEL attribute so that it can assume as its value a label constant at some other point in the program.

language character set. A set of characters that has been defined to represent program elements in the source language.

leading zeroes. Zeros that have no significance in the value of an arithmetic integer. All zeros to the left of the first significant integer digit of a number.

level-number. A number that precedes a name in a DECLARE statement and specifies the organization of the structure in that statement.

level-one variable. A major structure name. Any unsubscripted variable not contained within a structure.

lexically. Relating to the left-to-right order of units.

list-directed. The type of stream-oriented transmission in which data in the stream appears as constants separated by blanks or commas and for which formatting is provided automatically.

locator. A type of variable that identifies a location in storage.

locator qualification. In a reference to a based variable, either a locator variable or function reference connected by an arrow to the left of a based variable to specify the generation of the based variable to which the reference refers, or the implicit connection of a locator variable with the based reference.

locator value. A value that identifies the storage address.

locator variable. A variable whose value identifies the location in main storage of a variable or a buffer. It has the POINTER or OFFSET attribute.

locked record. A record in an EXCLUSIVE DIRECT UPDATE file that is available to only one task at a time.

logical level (of a structure member). The depth indicated by a level number when all level numbers are in direct sequence (when the increment between successive level numbers is one).

logical operators. The bit-string operators ¬ (not), & (and), and | (or).

loop. A sequence of instructions that is executed iteratively.

lower bound. The lower limit of an array dimension.

M

main procedure. An external procedure whose PROCEDURE statement has the OPTIONS (MAIN) attribute. This procedure is invoked automatically as the first step in the execution of a program.

major structure. A structure whose name is declared with level number 1.

manifest declaration. The declaration of a name in a DECLARE statement.

member. A subdivision of a structure consisting either of a minor structure or of a scalar.

minor structure. A structure that is contained within another structure. The name of a minor structure is declared with a level number greater than one.

mode (of arithmetic data). An attribute of arithmetic data—it is either *real* or *complex*.

multiple declaration. (1) Two or more declarations of the same identifier internal to the same block without different qualifications. (2) Two or more external declarations of the same identifier with different attributes in the same program.

multiprocessing. The use of a computing system with two or more processing units to execute two or more programs simultaneously.

multiprogramming. The use of a computing system to execute more than one program concurrently, using a single processing unit.

multitasking. A facility that allows a programmer to execute more than one PL/I procedure simultaneously.

N

name. Any identifier that the user assigns to a variable or to a constant. An identifier appearing in a context where it is not a keyword. Sometimes called a user-defined name.

nesting. The occurrence of:

- A block within another block
- A group within another group
- An IF statement in a THEN clause or in an ELSE clause
- A function reference as an argument of a function reference
- A remote format item in the format list of a FORMAT statement
- A parameter descriptor list in another parameter descriptor list
- An attribute specification within a parenthesized name list for which one or more attributes are being factored.

non-connected storage. Separate locations in storage containing related items of data that can be referred to by a single name but that are separated by other data items not referred to by that name.

normal default. The default for an attribute (or set of attributes) that occurs in the absence of one or more DEFAULT statements in a program.

nonprintable types. A type describing data that cannot be transmitted using the PUT LIST statement. Nonprintable types are all scalar types including the graphic type, other than the printable scalar types.

null locator value. A special locator value that cannot identify any location in internal storage. It gives a positive indication that a locator variable does not currently identify any generation of data.

null statement. A statement that contains only the semicolon symbol (;). It indicates that no action is to be taken.

null string. A character or bit string with a length of zero.

numeric-character data. See *decimal picture data*.

O

object. A collection of data referred to by a single name.

offset variable. A locator variable with the OFFSET attribute, whose value identifies a location in storage relative to the beginning of an area.

ON-condition. An occurrence, within a PL/I program, that could cause a program interrupt. It may be the detection of an unexpected error or of an occurrence that is expected, but at an unpredictable time.

ON-statement action. The action explicitly established for a condition when the condition is raised. The ON-statement action overrides or suspends any previously established action unless it is overridden by a further ON-statement for the same condition or the

block it was processed in ends. Contrast with *implicit action*.

on-unit. The specified action to be executed on detection of the enabled condition named in the containing ON statement. This excludes SYSTEM and SNAP.

opening (of a file). The association of a file with a data set and the completion of a full set of attributes for the file name.

operand. An expression to whose value an operator is applied.

operational expression. An expression that consists of one or more operators.

operator. A symbol specifying an operation to be performed.

option. A specification in a statement that may be used to influence the execution or interpretation of the statement.

P

packed decimal. The internal representation of a fixed-point decimal data item.

padding. (1) One or more characters or bits concatenated to the right of a string to extend the string to a required length. For character strings, padding is with blanks. For bit strings, padding is with zeros. (2) One or more characters or bits inserted in a structure so that the structure elements have the required alignment.

parameter. A name in a procedure that is represents an argument passed to that procedure.

parameter descriptor. The set of attributes specified for a single parameter in an ENTRY attribute specification.

parameter descriptor list. The list of all parameter descriptors in an ENTRY attribute specification.

parameter list. A parenthesized list of one or more parameters, separated by commas and following either the keyword PROCEDURE in a procedure statement or the keyword ENTRY in an ENTRY statement. The list corresponds to a list of arguments passed at invocation.

partially-qualified name. A qualified name that is incomplete. It includes one or more, but not all, of the names in the hierarchical sequence above the structure member to which the name refers, as well as the name of the member itself.

path testing. Debugging technique that involves selecting test data that will enable the testing of all parts of a program.

picture data. Arithmetic data represented in character form.

picture specification. A data item that has a numeric value but that can also be represented as a character value according to the editing characters specified in the item's declaration.

picture specification character. Any of the characters that can be used in a picture specification.

point of invocation. The point in the invoking block at which the procedure reference to the invoked procedure appears.

pointer. A type of variable that identifies a location in storage.

pointer value. A value that identifies the storage address.

pointer variable. A locator variable with the POINTER attribute, whose value identifies an absolute location in main storage.

precision. The number of digits contained in a fixed-point data item, or the minimum number of significant digits (excluding the exponent) maintained for a floating-point data item.

prefix. A label or a parenthesized list of one or more condition names included at the beginning of a statement.

prefix operator. An operator that precedes an operand and applies only to that operand. The prefix operators are + (plus), – (minus), and ¬ (not).

preprocessor. A program that examines the source program for preprocessor statements which are then executed, resulting in the alteration of the source program.

preprocessor statement. A special statement appearing in the source program that specifies how the source program text is to be altered. It is executed as it is encountered by the preprocessor.

problem data. Coded arithmetic, bit, character, and picture data that represents values processed by the program.

procedure. A collection of statements, delimited by PROCEDURE and END statements. A procedure is a program or a part of a program, delimits the scope of names, and is activated by a reference to its entry

name. See also *external procedure* and *internal procedure*.

procedure reference. An entry constant or variable or a built-in function name. The variable may be followed by one or more argument lists. It may appear in a CALL statement or CALL option, or as a function reference.

processor. A program that prepares source program text for execution.

program. A set of one or more external procedures, one of which must have the OPTIONS(MAIN) specification in its procedure statement.

program control data. Area, locator, label, format, entry, and file data that is used to control the processing of a PL/I program.

printable type. A type describing data that can be transmitted using the PUT LIST statement. Printable types are the numeric types, the bit and character types, and the pictured types.

prologue. The processes that occur automatically on block activation.

pseudovariable. Any of the built-in function names that can be used to specify a target variable.

pseudovariable reference. A value reference which has a pseudovariable as its object.

Q

qualified name. A hierarchical sequence of names of structure members, connected by periods, used to identify a component of a structure. Any of the names may be subscripted.

R

range (of a default specification). A set of identifiers and/or parameter descriptors to which the attributes in a default specification of a DEFAULT statement apply.

record. The logical unit of transmission in a record-oriented input or output operation.

recorded key. An identifier within a set of data elements recorded in a direct-access volume to identify an associated data record.

record-oriented data transmission. The transmission of data in the form of separate records. Contrast with *stream data transmission*.

recursive procedure. An active procedure that can be called from within itself or from within another active procedure.

reentrant procedure. A procedure that can be reactivated while it is still active in another active procedure.

REFER expression. The expression preceding the keyword REFER, from which an original bound, length, or size is taken when a based variable containing a REFER option is allocated, either by an ALLOCATE or LOCATE statement.

REFER object. The variable in a REFER option that specifies the current bound, length, or size for a member of a based structure. The REFER object must be a member of the structure. It must not be locator-qualified or subscripted, and it must precede the member declared with the REFER option.

reference. The appearance of a name, except in a context that causes explicit declaration.

remote format item. The letter R specified in a format list together with the label of a separate format statement. The format statement is used by edit-directed data transmission statements to control the format of data being transmitted.

repetition factor. A parenthesized unsigned decimal integer constant that specifies:

1. The number of occurrences of a string configuration that make up a string constant.
2. The number of occurrences of a picture specification character in a picture specification.

repetitive specification. An element of a data list that specifies controlled iteration to transmit one or more data items, generally used in conjunction with arrays.

restricted expression. An expression whose value is calculated at compile-time and used as a constant.

returned value. The value returned by a function procedure to the point of invocation.

returns descriptor. A descriptor used in a RETURNS attribute, and in the RETURNS option of the PROCEDURE and ENTRY statements.

run unit. A set of PL/I programs, each of which is called by some other PL/I program within the set, except for the initially called program, which is called from outside the set. A PL/I run unit is suspended when a program in the run unit calls a non-PL/I program, and is resumed when the called program returns control to the PL/I program that called it. A PL/I run unit ends when the initially called PL/I program returns control to the non-PL/I program that originally called the initial program and so started the run unit.

S

scalar. A type of program object that contains either string or numeric data. It provides the byte string it is mapped to with representation and operational characteristics. Contrast with *pointer*.

scalar item. A single item of data; an element.

scalar type. The type describing scalar data.

scalar variable. A variable that represents a single data item.

scale. A system of mathematical notation whose representation of an arithmetic value is either fixed-point or floating-point.

scale factor. A specification of the number of fractional digits in a fixed-point number.

scope (of a condition prefix). The portion of a program throughout which a particular condition prefix applies.

scope (of a declaration). The portion of a program throughout which the meaning of a particular name does not change.

scope (of a name). The portion of a program throughout which the meaning of a particular name does not change.

select-group. A sequence of selection clauses delimited by SELECT and END statements. The select-group is used for control purposes.

selection clause. A WHEN or OTHERWISE clause of a select-group.

self-defining data. A data item, or an aggregate of data items, that includes descriptive information about the attributes of the data, such as values for adjustable bounds or lengths.

separator. See *delimiter*.

sign and currency symbol characters. The picture specification characters. S, +, –, and $ (or other national currency symbols). They can be used:

- As static characters in which case they are specified only once in a picture specification and appear in the associated data item in the position in which they have been specified.
- As drifting characters, in which case they are specified more than once (as a string in a picture specification) but appear in the associated data item at most once, immediately to the left of the significant portion of the data item.

significant allocation. Any unfreed allocation in an area and any freed allocation that lies between the start of the area and the end of the unfreed allocation that is farthest from the start of the area. If a subsequent allocation of the same size is made in the same allocation, the original allocation ceases to be significant.

simple statement. See *statement body*.

source key. A key referred to in a record-oriented transmission statement that identifies a particular record within a direct-access data set.

source program. A program that serves as input to the compiler. The source program may contain preprocessor statements.

source variable. A variable whose value is to be assigned or to take part in some other operation.

standard default. The alternative attribute or option assumed when none has been specified and there is no applicable DEFAULT statement.

standard file. A file assumed by the processor in the absence of a FILE or STRING option in a GET or PUT statement. SYSIN is the standard input file and SYSPRINT is the standard output file.

standard system action. Action specified by the language to be taken for an enabled condition in the absence of an on-unit for that condition.

statement. A grouping of identifiers, constants, and delimiters that makes up do groups and blocks. The end of a statement is indicated by a semicolon (;). See also *keyword statement, assignment statement*, and *null statement*.

statement body. A statement body can be either a simple or a compound statement. (1) A simple statement is a statement with a simple body. There are three types of simple statements: keyword, assignment and null. Each type contains a statement body that is terminated by a semicolon. (2) A compound statement begins with a keyword. All compound statements are keyword statements. There are four compound statements: IF, ON, WHEN, and OTHERWISE. A compound statement is terminated by a semicolon that also terminates the statement body.

statement identifier. The PL/I keyword that indicates the purpose of a statement.

statement-label constant. See *label constant*.

statement-label expression. See *label expression*.

statement-label variable. See *label variable*.

static storage allocation. The allocation of storage for static variables.

static variable. A variable that is allocated before execution of the program begins and that remains allocated for the duration of execution.

stream-oriented data transmission. The transmission of data in which the organization of the data into records is ignored and the data is treated as though it were a continuous stream of individual data values in character form. Contrast with *record-oriented data transmission*.

string. (1) A series of things, such as characters, in a line. (2) In PL/I, a contiguous sequence of characters or bits that is treated as a single data item. (3) A group of auxiliary storage devices connected to a system. The order and location in which each device is connected to the system determines the physical address of the device.

string variable. A variable declared with the BIT or CHARACTER attribute, whose values can be either bit strings or character strings.

structure. A collection of data items that need not have identical attributes. Contrast with *array*.

structure expression. An expression whose evaluation yields a structure set of values.

structure of arrays. A structure containing arrays specified by declaring individual members names with the dimension attribute.

structure members. Any of the minor structures or elementary names in a structure.

structuring. The makeup of a structure, in terms of the number of members, the order in which they appear, their attributes, and their logical level (but not necessarily their names or declared level numbers).

structure variable. A variable that represents an aggregate of data items that might not have identical attributes. Contrast with *array variable* and *scalar variable*.

subfield (of a picture specification). That portion of a picture specification field that appears before or after a V picture specification character.

subroutine. A procedure that has no RETURNS option in the PROCEDURE statement. Contrast with *function*.

subroutine call. An entry reference that must represent a subroutine, followed by an optional and possibly empty argument list that appears in a CALL statement. Contrast with *function reference*.

subscript. An element expression that specifies a position within a dimension of an array. A subscript can also be an asterisk, in which case it specifies the entire extent of the dimension.

subscript list. A parenthesized list of one or more subscripts, one for each dimension of an array, which together uniquely identify either a single element or cross section of the array.

synchronous. Using a single flow of control for serial execution of a program.

T

target reference. A reference that designates a generation to receive an item of data.

target variable. A variable to which a value is assigned.

termination (of a block). Cessation of execution of a block, and the return of control to the activating block by means of a RETURN or END statement, or the transfer of control to the activating block or to some other active block by means of a GO TO statement.

truncation. The removal of one or more digits, characters, or bits from one end of an item of data when a string length or precision of a target variable has been exceeded.

type. The set of data attributes and storage attributes that apply to a generation, a value, or an item of data.

U

unconnected aggregate. See *inherited dimensions*.

undefined. Indicates something that is not defined by the language and that may change without notice. Thus, programs that seem to work correctly when referencing undefined results do so by chance and are in error.

union. A set of variants.

upper bound. The upper limit of an array dimension.

V

value reference. A reference used to obtain the value of an item of data.

variable. A named entity used to refer to data and to which values can be assigned. Its attributes remain constant, but it can refer to different values at different times.

variable reference. A reference that designates all or part of a variable.

virtual point picture character. The V picture specification character, which is used in picture specifications to indicate the position of an assumed decimal or binary point.

Z

zero-suppression characters. The picture specification characters Z and *, which are used to suppress zeros in the corresponding digit positions and replace them with blanks or asterisks respectively.

Index

Special Characters

Numerics

A

B

C

D

E

F

G

H

I

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

Y

Z

Readers' Comments

IBM SAA AD/Cycle PL/I Package/2
Language Reference
Release 1

Publication No. SC26-4823-00

Please use this form to tell us what you think about the accuracy, clarity, organization, and appearance of this manual. Any suggestions you have for its improvement will be welcome.

In sending information to IBM, you grant IBM a non-exclusive right to use or distribute the information in any way it believes appropriate without incurring any obligation to you.

Note: Do not use this form to request IBM publications. Please direct any requests for copies of publications, or for assistance in using your IBM system, to your IBM representative or to the IBM branch office serving your locality.

Mail the completed form to the address on the reverse side. If you mail it from a country other than the United States, give it to your IBM representative or to the IBM branch office serving your locality for postage-paid mailing.

If you prefer to send comments by fax, use this U.S. number: (408) 463-4393

If you would like a reply, be sure to print your name, and your address or phone number, below.

Name _____ Address _____

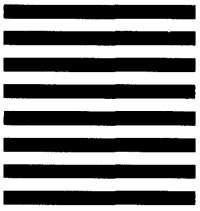
Company or Organization _____ _____

Phone No. _____ _____

IBM®

Fold and Tape | **Please do not staple** | Fold and Tape

NO POSTAGE
NECESSARY
IF MAILED IN THE
UNITED STATES



BUSINESS REPLY MAIL

FIRST CLASS MAIL PERMIT NO. 40 ARMONK, NEW YORK

POSTAGE WILL BE PAID BY ADDRESSEE

International Business Machines Corporation
Department J58
PO BOX 49023
SAN JOSE CA 95161-9945

Fold and Tape | **Please do not staple** | Fold and Tape

Readers' Comments

IBM SAA AD/Cycle PL/i Package/2
Language Reference
Release 1

Publication No. SC26-4823-00

Please use this form to tell us what you think about the accuracy, clarity, organization, and appearance of this manual. Any suggestions you have for its improvement will be welcome.

In sending information to IBM, you grant IBM a non-exclusive right to use or distribute the information in any way it believes appropriate without incurring any obligation to you.

Note: Do not use this form to request IBM publications. Please direct any requests for copies of publications, or for assistance in using your IBM system, to your IBM representative or to the IBM branch office serving your locality.

Mail the completed form to the address on the reverse side. If you mail it from a country other than the United States, give it to your IBM representative or to the IBM branch office serving your locality for postage-paid mailing.

If you prefer to send comments by fax, use this U.S. number: (408) 463-4393

If you would like a reply, be sure to print your name, and your address or phone number, below.

Name _____ Address _____

Company or Organization _____ _____

Phone No. _____ _____

IBM®

Cu
Alc

Fold and Tape **Please do not staple** Fold and Tape

NO POSTAGE
NECESSARY
IF MAILED IN THE
UNITED STATES

BUSINESS REPLY MAIL

FIRST CLASS MAIL PERMIT NO. 40 ARMONK, NEW YORK

POSTAGE WILL BE PAID BY ADDRESSEE

International Business Machines Corporation
Department J58
PO BOX 49023
SAN JOSE CA 95161-9945

Fold and Tape **Please do not staple** Fold and Tape

Cut
Alo

Readers' Comments

IBM SAA AD/Cycle PL/I Package/2
Language Reference
Release 1

Publication No. SC26-4823-00

Please use this form to tell us what you think about the accuracy, clarity, organization, and appearance of this manual. Any suggestions you have for its improvement will be welcome.

In sending information to IBM, you grant IBM a non-exclusive right to use or distribute the information in any way it believes appropriate without incurring any obligation to you.

Note: Do not use this form to request IBM publications. Please direct any requests for copies of publications, or for assistance in using your IBM system, to your IBM representative or to the IBM branch office serving your locality.

Mail the completed form to the address on the reverse side. If you mail it from a country other than the United States, give it to your IBM representative or to the IBM branch office serving your locality for postage-paid mailing.

If you prefer to send comments by fax, use this U.S. number: (408) 463-4393

If you would like a reply, be sure to print your name, and your address or phone number, below.

Name _____ Address _____

Company or Organization _____ _____

Phone No. _____ _____

IBM®

Fold and Tape **Please do not staple** Fold and Tape

NO POSTAGE
NECESSARY
IF MAILED IN THE
UNITED STATES

BUSINESS REPLY MAIL

FIRST CLASS MAIL PERMIT NO. 40 ARMONK, NEW YORK

POSTAGE WILL BE PAID BY ADDRESSEE

International Business Machines Corporation
Department J58
PO BOX 49023
SAN JOSE CA 95161-9945

Fold and Tape **Please do not staple** Fold and Tape

Readers' Comments

IBM SAA AD/Cycle PL/I Package/2
Language Reference
Release 1

Publication No. SC26-4823-00

Please use this form to tell us what you think about the accuracy, clarity, organization, and appearance of this manual. Any suggestions you have for its improvement will be welcome.

In sending information to IBM, you grant IBM a non-exclusive right to use or distribute the information in any way it believes appropriate without incurring any obligation to you.

Note: Do not use this form to request IBM publications. Please direct any requests for copies of publications, or for assistance in using your IBM system, to your IBM representative or to the IBM branch office serving your locality.

Mail the completed form to the address on the reverse side. If you mail it from a country other than the United States, give it to your IBM representative or to the IBM branch office serving your locality for postage-paid mailing.

If you prefer to send comments by fax, use this U.S. number: (408) 463-4393

If you would like a reply, be sure to print your name, and your address or phone number, below.

Name

Address

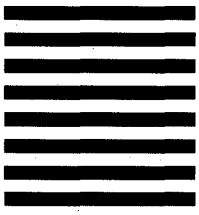
Company or Organization

Phone No.

Cu
Alc

Fold and Tape **Please do not staple** Fold and Tape

NO POSTAGE
NECESSARY
IF MAILED IN THE
UNITED STATES



BUSINESS REPLY MAIL

FIRST CLASS MAIL PERMIT NO. 40 ARMONK, NEW YORK

POSTAGE WILL BE PAID BY ADDRESSEE

International Business Machines Corporation
Department J58
PO BOX 49023
SAN JOSE CA 95161-9945

Fold and Tape **Please do not staple** Fold and Tape

Cut
Alor

IBM®

Program Number: 5601-388

Printed in U.S.A. on Recycled Paper

PL/I Package/2 Library

GC26-3090	Fact Sheet
GC26-4821	Licensed Program Specifications
SX26-3822	Installation
SC26-4822	Programming Guide
SC26-4823	Language Reference
SX26-3793	Reference Summary
SC26-3133	Language Environment Run-Time Messages

SC26-4823-00