IBM PL/I for VSE/ESA

Language Reference

Release 1

SC26-8054-00

IBM PL/I for VSE/ESA

Language Reference

Release 1

SC26-8054-00

Note!

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

First Edition (April 1995)

This edition applies to Version 1 Release 1 of IBM PL/I for VSE/ESA, 5686-069, 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 Licensing, IBM Corporation, 500 Columbus Avenue, Thornwood, NY 10594, U.S.A.

Programming Interface Information

This book is intended to help the customer write programs using IBM PL/I for VSE/ESA. This book documents General-use Programming Interface and Associated Guidance Information provided by IBM PL/I for VSE/ESA.

General-use programming interfaces allow the customer to write programs that obtain the services of IBM PL/I for VSE/ESA.

Trademarks

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

AD/Cycle
BookManager
CICS
CICS/VSE
IBM
Language Environment
OS/2
SAA
VSE/ESA

About this book

This book is a reference for the PL/I programmer. 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 IBM PL/I for VSE/ESA (PL/I VSE).

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

Note: In this book, unless otherwise stated, the term VSE means VSE/ESA*.

Using your documentation

The publications provided with PL/I VSE are designed to help you do PL/I programming under VSE. Each publication helps you perform a different task.

Where to look for more information

For information about the PL/I VSE library, see Table 1.

Table 1. How to use the publications you receive with PL/I VSE

To...	Use...
Evaluate the product	*Fact Sheet*
Understand warranty information	*Licensed Program Specifications*
Install the compiler	*Installation and Customization Guide*
Understand product changes and adapt programs to PL/I VSE	*Migration Guide*
Prepare and test your programs and get details on compiler options	*Programming Guide*
Get details on PL/I syntax and specifications of language elements	*Language Reference* *Reference Summary*
Diagnose compiler problems and report them to IBM	*Diagnosis Guide*
Get details on compile-time messages[1]	*Compile-Time Messages and Codes*

Note:

1. For details on run-time messages, see the LE/VSE library.

You might also require information about IBM* Language Environment* for VSE/ESA* (LE/VSE). For information about the LE/VSE library, see Table 2.

Table 2 (Page 1 of 2). How to use the publications you receive with LE/VSE

To...	Use...
Evaluate Language Environment	*Fact Sheet* *Concepts Guide*
Install LE/VSE	*Installation and Customization Guide*
Understand the LE/VSE program models and concepts	*Concepts Guide* *Programming Guide*

Table 2 (Page 2 of 2). How to use the publications you receive with LE/VSE

To...	Use...
Prepare your LE/VSE-conforming applications and find syntax for run-time options and callable services	*Programming Guide* *Reference Summary*
Debug your LE/VSE-conforming application and get details on run-time messages	*Debugging Guide and Run-Time Messages*
Diagnose problems that occur in your LE/VSE-conforming application	*Diagnosis Guide*
Understand warranty information	*Licensed Program Specifications*

For the complete titles and order numbers of these and other related publications, see the "Bibliography" on page 437.

What is new in PL/I VSE

This is a major new release of PL/I, containing many new features and facilities. It brings to VSE many of the functions of the MVS & VM version of PL/I (IBM SAA* AD/Cycle* PL/I MVS & VM), while retaining close source compatibility with the DOS PL/I Optimizing Compiler (DOS PL/I).

PL/I VSE enables you to integrate your PL/I applications into IBM Language Environment for VSE/ESA (LE/VSE). In addition to PL/I's already impressive features, you gain access to LE/VSE's rich set of library routines and enhanced interlanguage communication (ILC) with IBM COBOL for VSE/ESA (COBOL/VSE).

IBM Language Environment for VSE/ESA support

PL/I VSE provides the following functions in the LE/VSE area:

Interlanguage communication (ILC) support:

- Object code produced by PL/I VSE Release 1 can be linked with object code produced by other LE/VSE-conforming compilers (currently only COBOL/VSE).
- PL/I VSE programs can fetch COBOL/VSE phases.
- COBOL/VSE programs can fetch PL/I VSE phases.

Note: PL/I VSE does not support ILC with:

- FORTRAN
- RPG
- DOS/VS COBOL
- C/370*

Limited ILC support is provided for VS COBOL II at Release 3.2 or later.

Common support for multiple operating environments:

- Some of the restrictions on PL/I coding in the CICS* environment have been lifted.
- Procedure OPTIONS option FETCHABLE can be used to specify the procedure that gets control within a fetched phase.
- CEETDLI is supported in addition to PLITDLI and EXEC DLI.

- LE/VSE services provide storage management and condition handling support, as well as PLIDUMP and MSGFILE support for PL/I messages and other output.
- By default, only user-generated output is written to SYSLST. All run-time generated messages are written to MSGFILE.
- ERROR conditions now get control of all system abends. The PL/I message is issued only if there is no ERROR on-unit or if the ERROR on-unit does not recover from the condition via a GOTO.
- Selected items from PL/I Package/2 (the PL/I product for OS/2*) are implemented to allow better coexistence.
  - Limited support of OPTIONS(BYVALUE and BYADDR)
  - Limited support of EXTERNAL(environment-name) allowing alternate external names
  - Limited support of OPTIONAL arguments/parameters
  - Support for %PROCESS statement
  - NOT and OR compiler options

Product packaging:

- All PL/I VSE resident library routines are now packaged with LE/VSE, and are loaded at run time rather than link-edited with the application program. Changes to the resident library no longer require PL/I programs to be re-linked.
- At link-edit time, you have the option of getting math results that are compatible with LE/VSE or with DOS PL/I.
- Installation enhancements are provided to ease product installation and migration.

For migration considerations, see the *PL/I VSE Migration Guide.*

Usability enhancements

These enhancements expand the PL/I language statements and options, PL/I data types, and compiler options, to make the language easier to use.

Enhanced double-byte character set (DBCS) support: This support introduces many enhancements that facilitate processing of GRAPHIC and mixed-character data and allows the source of the PL/I program to be in DBCS and/or the single-byte character set (SBCS), rather than only in SBCS.

Hexadecimal data constants: Constants for bit and character data can now be defined in hexadecimal notation, such that each *character* (0-9 and A-F) represents 4 bits.

Interface improvements for all (sub)systems: A new compiler option, SYSTEM, lets the programmer specify the target operating environment (of the generated object code), and the format of the parameters for the MAIN procedure.

Specification of compile-time options: You can specify compile-time options on the *PROCESS statement, a new %PROCESS statement, and in the PARM option of the EXEC IEL1AA JCL statement.

Linking after errors: The COMPILE compile-time option has been enhanced to allow linking to proceed after a severe error.

Run-time options: You can specify program run-time options in the PARM option of the EXEC JCL statement. PL/I VSE and LE/VSE will use these to control the execution of PL/I programs.

Passing parameters to the MAIN procedure: VSE JCL can also be used to pass a parameter to the MAIN PL/I procedure. A slash (/) separates the run-time options from the program parameter.

OPEN statement enhancements:

- There are new parameters on the PL/I OPEN statement that allow additional file attributes to be specified at file open time. These attributes are added to those in the file declaration.
- A vendor exit on the PL/I OPEN statement can be used to change the system logical unit number of the PL/I spill file.
- Data set name sharing for VSAM files, using the DSN option of the ENVIRONMENT attribute.

Date and time enhancements: A new built-in function, DATETIME, returns consistent date and time, including the four-digit year.

PL/I statement numbering options: A new compiler option, NUMBER, specifies that PL/I statement numbers will be derived from the sequence numbers in the program source deck, instead of being allocated sequentially.

Dynamic loading of external procedures: PL/I now supports the FETCH and RELEASE statements, to load external procedures into main storage at run time instead of having them link-edited with the MAIN procedure. (If these external procedures are PL/I, they must be compiled with PL/I VSE.)

New I/O facilities: PL/I VSE provides the following new I/O facilities:

- Support for REGIONAL(2) files
- Support for V and VS formats on REGIONAL(3) files
- Support for DELETE statement on REGIONAL files
- Support for multitrack search on REGIONAL files, using the LIMCT option of the ENVIRONMENT attribute
- Support for VSAM variable-length relative-record data sets (VRDS)
- Support for V format on consecutive unbuffered files
- Support for VS and VBS formats on consecutive buffered files

System programmer functions: A number of significant new features enhance PL/I as a system programming language:

- Support of additional program execution environments.

  PL/I can now be used for some system exit routines, such as the LE/VSE initialization exit.
- Additional support for pointers.

PL/I built-in functions are now available to perform extended operations on pointers, including pointer arithmetic.

- Additional support for entry variables.

  A new built-in function and pseudovariable, ENTRYADDR, allows programmers to manipulate entry point addresses of procedures.

Extended addressing enhancements

These enhancements exploit the large amounts of storage available in the VSE/ESA environment, making programming easier.

Addressing mode: PL/I VSE programs can be link-edited with AMODE(31) and RMODE(ANY).

Location of variables: PL/I variables can now be located above the 16-megabyte line.

Fullword array subscripts: Array bounds can now be in the range -2^{31} (-2,147,483,648) through $+2^{31}-1$ (+2,147,483,647). The associated built-in functions (such as LBOUND and HBOUND) now return FIXED BINARY(31) values.

AREA and aggregate sizes: An AREA can now have a maximum size of 2,147,483,647 ($2^{31}-1$) bytes.

An aggregate can now have a maximum size of 2,147,483,647 ($2^{31}-1$) bytes. For unaligned BIT arrays and aggregates that contain any unaligned BIT data (arrays or non-arrays), the maximum size is 268,435,455 ($2^{28}-1$) bytes.

These numbers include any control information bytes that might be needed.

Notation conventions used in this book

This section describes how information is presented in this book. Examples are presented in all uppercase letters. User-supplied information is presented in all lowercase letters.

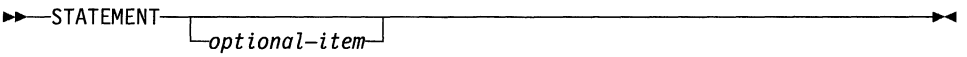
Throughout this book, syntax is described using the following structure:

- 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.



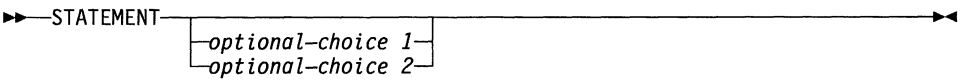
- When you can choose from two or more items, the items appear vertically, in a stack. If you **must** choose one of the items, one item of the stack appears on the main path. The default, if any, appears above the main path and is chosen by the compiler if you do not specify another choice.

  Note: In some cases, the default is affected by the:

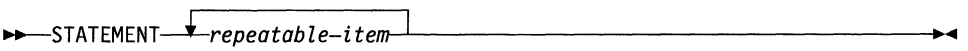
  - System in which the program is being run
  - Environmental parameters specified

  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.

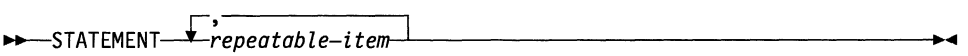


- An arrow returning to the left above the main line is a *repeat arrow*, and it indicates an item that can be repeated.



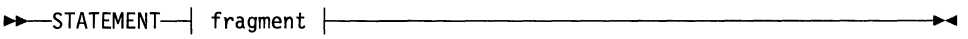
  A repeat arrow above a stack indicates that you can make more than one choice from the stacked items, or repeat a single choice.

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

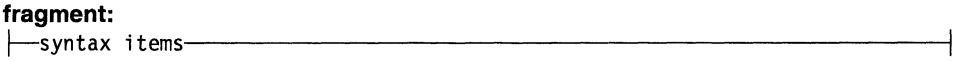


  If the comma appears below the repeat arrow line instead of on the line as shown in the previous example, the comma is optional as a separator of items in a series.

- 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.



  fragment:



- 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 something is *undefined*, it is not part of the language and may change at any time. The word "undefined" does not mean that we (or you) cannot describe what the current implementations do. Programs that happen to "work" when using undefined results are 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 a statement.
- The lowercase letter b, when not in a word, indicates a blank character.

Industry standards used

PL/I VSE 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).

Chapter 1. Program elements

Chapter 1. Program elements

This chapter describes PL/I program elements:

- Character sets recognized by PL/I
  - Single-byte character set (SBCS)
  - Double-byte character set (DBCS)
- Statement elements
  - Identifiers
  - Delimiters
- Statements
- Groups

Note: The implementation limits for PL/I language elements are listed in Appendix, "PL/I limits" on page 434.

Character sets

PL/I programs contain characters from the single-byte character set (SBCS), the double-byte character set (DBCS), or both. National languages that use extensive character sets, such as Japanese, use DBCS.

Single-byte character set (SBCS)

Constants and comments can contain any SBCS character (that is, any EBCDIC value).

PL/I statements contain only:

- Alphanumeric characters (alphabetic characters and digits)
- "Special" characters

Alphabetic characters

There are 29 alphabetic characters: the 26 characters of the English alphabet (A-Z) and three *extralingual* characters ($, @, and #).

The following table shows the three extralingual characters with their equivalent EBCDIC and ASCII hexadecimal (hex) values.

Table 3. Extralingual equivalents

Character	Meaning	EBCDIC hex value	ASCII hex value
$	local currency symbol	5B	24
#	number sign	7B	23
@	commercial "at" sign	7C	40

Note: Code points for these symbols may vary between code pages.

The following table shows the characters of the English alphabet and their equivalent EBCDIC and ASCII hex values.

Table 4. 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
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

Digits

The following table shows the 10 decimal digits (0-9) and their equivalent EBCDIC and ASCII hex values.

Table 5 (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
7	F7	37

Table 5 (Page 2 of 2). Decimal digit equivalents

Character	EBCDIC hex value	ASCII hex value
8	F8	38
9	F9	39

"Special" characters

PL/I recognizes 21 special characters. The following table shows the special characters, their meanings, and their equivalent EBCDIC and ASCII hex values.

Table 6. 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
%	Percent symbol	6C	25
;	Semicolon	5E	3B
:	Colon	7A	3A
¬	NOT symbol	5F	5E
&	AND symbol	50	26
|	OR symbol	4F	7C
>	Greater than symbol	6E	3E
<	Less than symbol	4C	3C
_	Break character (underscore)	6D	5F
?	Question mark	6F	3F

Composite symbols

You can combine special characters to create composite symbols. The following table describes composite symbols and their meanings. Composite symbols cannot contain blanks ('40'X).

Table 7. Composite symbol description

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

Lowercase characters

You can use lowercase characters when writing a program. When used in keywords or names, the lowercase character converts to its corresponding uppercase character. This is true even if you entered the lowercase character as an EBCDIC DBCS character. When used in a comment or a character constant the lowercase character remains as lowercase.

See "Preprocessor scan" on page 413 for information on how the preprocessor handles lowercase characters.

Double-byte character set (DBCS)

This section describes the double-byte character set, shift control characters, and the DBCS blank.

Each DBCS character is represented by 2 bytes.

When using DBCS identifiers, the values '41'X through 'FE'X are allowed in either byte. If the first byte is '42'X, the second byte must be one of the hexadecimal values allowed in SBCS identifiers.

When using DBCS characters in a graphic constant, or in the DBCS portion of a mixed-character constant, PL/I allows the values '00'X through 'FF'X in either byte, except that it does not allow '0E'X and '0F'X in either byte. These two values are used as shift control characters (see the heading below).

"Using DBCS in source programs" on page 12 lists the language elements that allow DBCS characters, and the rules for using DBCS in PL/I programs.

Shift control characters

To use DBCS in a source program, you specify the GRAPHIC compiler option. This causes the compiler to interpret the values '0E'X and '0F'X as shift control characters.

Shift control characters identify the beginning and end of a DBCS portion in a source program. The shift-out character (SO) has a value of '0E'X and indicates that DBCS data begins in the next byte. The shift-in character (SI) has a value of '0F'X and indicates that DBCS data ends and that an SBCS character or shift-out character is next. This book uses the following symbols to represent shift control characters:

- < indicates the shift-out character '0E'X
- > indicates the shift-in character '0F'X

DBCS blank

The following table shows that the value '4040'X represents the DBCS blank character.

Table 8. DBCS blank equivalent

Character	Meaning	Hex value
bb	DBCS Blank	4040

Statement elements

This section describes the elements that make up a PL/I statement, using SBCS for source input. The section "Using DBCS in source programs" on page 12 describes how to use DBCS with these elements for source input.

A PL/I statement consists of identifiers, delimiters, constants, and iSUBs. Identifiers and delimiters are described here, constants are described in Chapter 2, "Data elements" on page 21, and iSUBs are described under "DEFINED attribute" on page 224.

Identifiers

An *identifier* is 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 alphabetic characters. The others, if any, can be alphabetic (including the three extralingual characters), digits, or the break character.

Identifiers can be keywords, programmer-defined names, or range specifications.

In the PL/I language, the compiler can determine from context if an identifier is a keyword. Hence you can use any identifier as a programmer-defined name. There are no *reserved* words in PL/I.

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. In creating a name, you must observe the syntax rules for creating an identifier. Any identifier can be used as a name.

At any point in a program, a name can have one and only one meaning. 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

FILE2                #32

LOOP_3
```

Additional requirements for programmer-defined external names are given in "INTERNAL and EXTERNAL attributes" on page 155.

PL/I keywords

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

Delimiters and operators

Delimiters and *operators* are used to separate identifiers, iSUBs, and constants. (The PICTURE keyword does not need to be separated from its picture specification.) Table 9 shows delimiters; Table 10 shows operators.

Table 9 (Page 1 of 2). 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 point
semicolon	;	Terminates statements
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 of a statement
parentheses	( )	Enclose lists, expressions, iteration factors, and repetition factors; enclose information associated with various keywords
pointer	->	Denotes locator qualification

Table 9 (Page 2 of 2). Delimiters

Name	Delimiter	Use
percent symbol	%	Indicates % statements and preprocessor statements
asterisk	*	Indicates *PROCESS statement
single quote	'	Encloses constants (indicates the beginning and end of a constant)

Note: Inadvertent omission of certain symbols can cause errors that are difficult to trace. Common errors are: unbalanced quotes, unmatched parentheses, unmatched comment delimiters (for example, /* instead of */ when closing a comment), and missing semicolons.

Table 10. Operators

Operator type	Character(s)	Description
arithmetic operators	+ - * / **	Addition or prefix plus Subtraction or prefix minus Multiplication Division Exponentiation
comparison operators	> ¬> >= = ¬= <= < ¬<	Greater than Not greater than Greater than or equal to Equal to Not equal to Less than or equal to Less than Not less than
bit operators	¬ & |	NOT AND OR
string operator	||	Concatenation

The characters used for delimiters can be used in other contexts. For example, the period is a delimiter when used in name qualification, such as A.B. However, it is part of the constant when used in a decimal constant, such as 3.14.

Blanks

You can surround each operator or delimiter with blanks. 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, such as between keywords in a statement.

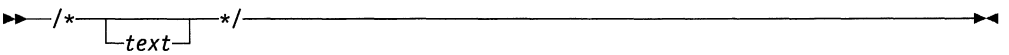
Blanks cannot occur within identifiers, iSUBs, composite symbols, or constants (except character, mixed, or graphic constants).

Other cases that require or allow blanks are noted in the text where the feature of the language is discussed. Some examples are:

```
AB+BC           is equivalent to      AB + BC
TABLE(10)       is equivalent to      TABLE ( 10   )
FIRST,SECOND    is equivalent to      FIRST, SECOND
ATOB            is not equivalent to  A TO B
```

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 by the program and do not affect execution 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. *Text* can be omitted. If you use the preprocessor, the values '00'X through '06'X must not be used.

***/** Specifies the end of a comment.

Here is an example of a comment within a statement:

```
A = /*  THIS SENTENCE COULD BE
        INSERTED AS A COMMENT  */  5;
```

The following example is a constant, rather than a comment, because it is enclosed in single quotes:

```
A = '/*   THIS IS A CONSTANT,
        NOT A COMMENT  */';
```

Statements

You use identifiers, constants, delimiters, and iSUBs to construct PL/I *statements*. The following description applies to your program after any preprocessing statements (see "%INCLUDE statement" on page 189) have been processed.

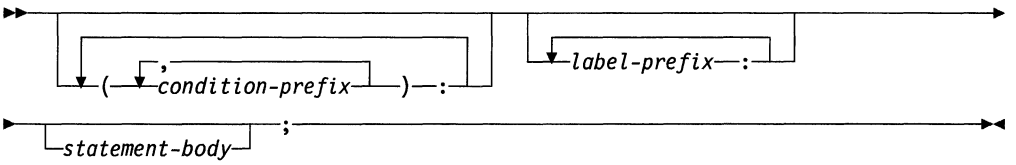
Although your program consists of a series of records or lines, the compiler views the program as a continuous stream. There are few restrictions in the format of PL/I statements, and programs can be written without considering special coding forms or checking to see that each statement begins in a specific column.

To avoid unexpected treatment of string constants that are continued, use the || (concatenate) operator. For example:

```
DECLARE     1...,
            2...INIT (('FIRST PART'
            ||         'SECOND PART'
            ||         'ETC')
                /*Note the extra parentheses
                inside INIT */),
            2...;
```

Each statement is terminated by a semicolon. A statement can begin in the next position after the previous statement, or it can be separated by any number of blanks.

Every statement must be contained within some enclosing group or block. The syntax for a PL/I statement is:



The following sections explain each part of the syntax.

Condition prefix

A *condition prefix* specifies the enabling or disabling of a PL/I condition. They are discussed in "Condition prefixes" on page 318. The following example enables the SIZE condition and disables the OVERFLOW condition:

```
(SIZE,NOOVERFLOW): COMPUTE: A=B*C**D;
```

This example can also be written as:

```
(SIZE):
(NOOVERFLOW):
COMPUTE:  A=B*C**D;
```

Any statement—except DECLARE, DEFAULT, ENTRY, or a % statement—can have one or more condition prefixes. In this book, syntax diagrams for individual statements generally do not show condition prefixes.

Label prefix

A *label prefix* is either a statement label or the initial value of an element of an array of non-STATIC label variables. Any statement—except WHEN and OTHERWISE statements and ON-units—can have one or more label prefixes. In this book, syntax diagrams for individual statements generally do not show label prefixes.

A statement label identifies a statement so that other statements (possibly in other programs) can transfer program control to the statement by referring to its label. Statement labels are either label-constants or entry-constants. ENTRY constants are sometimes referred to as *entry points*. (See "Label data and attribute" on page 39 and "Entry data" on page 133.)

The initial value of an element of an array of non-STATIC label variables is discussed under "INITIAL attribute" on page 230.

Statement body

There are two types of statement body: simple and compound. A *simple statement* is a statement with a simple body. A *compound statement* is a statement with a compound body.

Simple statements

There are three types of simple statements:

keyword statements Begin with a keyword indicating the function of the statement.

assignment statements Begin with an identifier and contain the assignment symbol (=).

null statements Null, except for labels and comments.

A semicolon terminates each simple statement (including null statements).

These are examples of simple statements:

```
READ FILE(IN) INTO(INPUT);          /* keyword statement      */

A = B + C;                          /* assignment statement   */

LABEL:;                             /* labeled null statement */

ALLOCATE A SET(P);                  /* keyword statement      */
```

Compound statements

Compound statements are *keyword statements*; they begin with a keyword indicating their function. Compound statements consist of one of four keywords —IF, ON, WHEN, or OTHERWISE—followed by one or more simple or compound statements. The semicolon terminating the final statement in a compound statement also terminates the compound statement.

These 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(''/* NULL */) ;
      OTHERWISE CALL OTHER_STMT;
    END;
    CALL PRINT;
  END;
```

Other statement types

There are two other types of statement: **%** statements and the ***** statement.

% statements

Statements beginning with a percent symbol (%) are either:

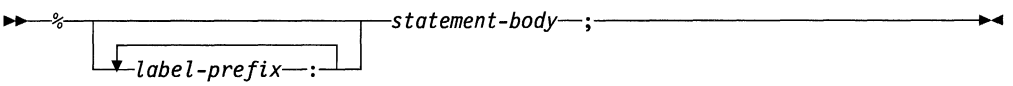
- PL/I statements directing the operation of the compiler (for example, to control compiler listings, or include program source text from a library). These are: %INCLUDE, %PRINT, %NOPRINT, %PAGE, %SKIP.

Or

- Preprocessor statements (described in Chapter 17, “Preprocessor facilities” on page 412).

Note: %INCLUDE can be either a PL/I statement or a preprocessor statement, depending on whether the INCLUDE or MACRO compiler option is specified (see the *PL/I VSE Programming Guide* for more information on compiler options).

The syntax for % statements is:



Label-prefix and *statement-body* are the same as for other statements except that % statements cannot have an element of an array of non-STATIC label variables as a label prefix.

* statement

The *PROCESS statement is the only PL/I statement beginning with an asterisk (*). The *PROCESS statement overrides compile-time options; it is a synonym for the %PROCESS statement.

For information on using the *PROCESS statement, see the *PL/I VSE Programming Guide*.

Groups

Statements can be contained within larger program units called groups. There are two types of group:

do-group A sequence of statements delimited by a DO statement and a corresponding END statement.

select-group A sequence of WHEN statements and an OTHERWISE statement delimited by a SELECT statement and a corresponding END statement.

The delimiting statements (DO and END, or SELECT and END) are part of the group.

When a group is used in a compound statement, control either flows into the group or bypasses it.

The flow of control within do-groups is described under "DO statement" on page 175; for select-groups, under "SELECT statement" on page 194.

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

Using DBCS in source programs

To use DBCS in a source program, you specify the GRAPHIC compiler option. This causes the compiler to interpret the values '0E'X and '0F'X as shift control characters, to shift out to DBCS mode and shift in to SBCS mode.

Each program record must begin and end in SBCS mode. If a statement containing a DBCS element continues on the next record, and the DBCS element is

at or near the end of the record, certain continuation rules apply. "DBCS continuation rules" on page 16 describes these rules.

A DBCS element or group of DBCS elements begins with a shift-out character (<) and ends with a shift-in character (>). Shift control characters must be in matching pairs within the record. Shift control character pairs cannot be nested. Each record can have any number of DBCS elements.

Some program elements do not support the use of DBCS. "Elements not supported by DBCS" on page 15 describes these elements.

DBCS identifiers

DBCS identifiers can be EBCDIC, non-EBCDIC, or a combination of both.

EBCDIC identifiers in DBCS form

EBCDIC DBCS identifiers must conform to the normal PL/I naming conventions, including the first-character rule. Using DBCS (two byte) form you can express any EBCDIC character by placing a '42'X in the first byte of each character, and the EBCDIC character in the second byte. An EBCDIC DBCS identifier is a synonym of the same identifier in SBCS. For example, in EBCDIC:

```
<.I.B.M> = 3;   is the same as   IBM = 3;
```

Note: This book uses the symbol "." (highlighted period) to represent the value '42'X.

Non-EBCDIC DBCS identifiers

DBCS names can be up to 14 DBCS characters in length. Names containing one or more non-EBCDIC DBCS characters are all DBCS. For example, in EBCDIC:

```
A<kk>B
A<kk.B>
<.Akk>B are all the same as <.Akk.B>
                (3 DBCS characters)
```

Note: This book uses "kk" to represent a non-EBCDIC DBCS character.

Uses for non-EBCDIC DBCS identifiers

Non-EBCDIC DBCS identifiers can be used for:

- Variable names of problem data or program control (area, entry, event, label, locator) data
- Internal procedure names and statement labels
- Programmer-defined internal condition names
- RANGE specification of a DEFAULT statement
- File names

Non-EBCDIC DBCS identifiers are not allowed for:

- EXTERNAL names, including outermost procedure names, secondary entries, and external conditions
- The TITLE value of the OPEN statement
- Procedure names in the FETCH and RELEASE statements
- The member name and type in the %INCLUDE statement

These names can be expressed in DBCS form, but must contain only EBCDIC ('42'X in the first byte) characters.

The I-through-N rule for language-specified defaults still applies. That is, identifiers beginning with non-EBCDIC DBCS characters are considered outside the I-through-N range. See "Language-specified defaults" on page 160 for more information.

DBCS language elements

The following list shows language elements that you can write using DBCS, with usage rules and examples:

Keywords and delimiters

Use SBCS, DBCS, or both.

```
DCL                     /* in SBCS                        */
<.D.C.L>                /* in DBCS                        */
<.D>C<.L>               /* in both SBCS and DBCS          */
```

Identifiers

Use SBCS, DBCS, or both.

```
DCL EOF                 /* in SBCS, is the same as        */
DCL <.E.O.F>            /* this in DBCS, is the same as   */
DCL <.E>O<.F>           /* this in both SBCS and DBCS     */

DCL <kkkk>X             /* these are all the same, where  */
DCL <kkkk.X>            /* kk is a valid, non-EBCDIC DBCS */
DCL <kkkk>x             /* character and x is the 3rd     */
DCL <kkkk.x>            /* character of the identifier.   */
```

Comments

Use SBCS, DBCS, or both.

Enclose in either SBCS or DBCS comment delimiters, with matching beginning and ending delimiters.

```
/* comment */                   /* all SBCS                      */
/* <.c.o.m.m.e.n.t> */          /* SBCS delimiters and DBCS text */
<./.*> comment <.*./>           /* DBCS delimiters and SBCS text */
<./.*   .c.o.m.m.e.n.t   .*./>  /* all DBCS                      */
```

Character, B, BX, B4, X, and GX constants

Enclose in either SBCS or DBCS quotes, with matching beginning and ending delimiters.

If a quote is part of the constant, and the same quote is the delimiter, the quote that is part of the constant must be doubled.

It is an error if the data in these constants contains DBCS with other than '42'X in the first byte.

Chapter 2, "Data elements" on page 21 gives a detailed description of each constant type.

```
'abc'                       /* all of these                 */
<.'.a.b.c.'>                /* are  'abc'                   */
'<.a.b>c'                   /*   3 characters               */

'ab''s'                     /* these are  'ab's'            */
<.'.a.b.'.'.s.'>            /*        4 characters          */

'c1c2'X                     /* these all                    */
'<.c.1.c.2>'X               /*          represent           */
<.'.C.1>C2<.'.X>            /*                 the same     */
'AB'                        /*                          data */
```

Mixed character constant

Enclose in either SBCS or DBCS quotes, with matching beginning and ending delimiters.

If a quote is required as part of the constant, and the same quote is the delimiter, the quote that is part of the constant must be doubled.

Data can be expressed in either SBCS or DBCS as required. The DBCS portion is not converted to SBCS, and adjacent SO/SI codes are retained.

The data is adjusted so that the DBCS portions are enclosed in shift codes.

```
'<.a.b.c>'M       stored as  <.a.b.c>       8 bytes
'<.I.B.M.'.S>'M   stored as  <.I.B.M.'.S>  12 bytes
'<.I.B.M>''<.S>'Mstored as  <.I.B.M>'<.S>13 bytes
'IBM<kk>'M        stored as  IBM<kk>        7 bytes
<.'>IBM<kk.'.M>   stored as  IBM<kk>        7 bytes
```

Graphic constant

Enclose in either SBCS or DBCS quotes, with matching beginning and ending delimiters.

If DBCS quotes enclose a constant, and a DBCS quote is part of the constant, the DBCS quote must be doubled.

Examples:

```
'<.a.b.c>'G                 /* a 6-byte-long Graphic constant  */
'<.I.B.M.'.S>'G             /* Graphic constant .I.B.M.'.S     */
<.'.I.B.M.'.'.S.'.G>        /* Graphic constant .I.B.M.'.S     */
```

Elements not supported by DBCS

Some language elements do not support DBCS. Using DBCS in these elements causes syntax errors. The following elements cannot contain DBCS characters, or are restricted as noted:

- Compiler options, when specified in either PARM or *PROCESS statements.
- The *PROCESS statement (including the *).
- The fields in the source record that are referred to by the MARGINS(m,n,c) compile-time option can be in DBCS. However:
  - The starting and ending columns are treated as single-byte characters.

  - The ASA control character is extracted from the specified column and treated as SBCS without regard to shift control characters. (Use the %SKIP and %PAGE statements.)
- The fields in the source record that are referred to by the SEQUENCE option, and are used when the NUMBER option is in effect, should be in SBCS. Also:
  - PL/I treats the starting and ending columns of SEQUENCE options as single-byte characters.
  - PL/I extracts sequence numbers from the specified columns, and treats them as SBCS numbers, ignoring shift control characters.
- You specify PL/I run-time options using only SBCS, except in the PLIXOPT declaration, where you can use DBCS for character string constants.

DBCS continuation rules

Each input record must begin and end in SBCS mode. When a DBCS element approaches the end of a record, some unique conditions can arise. Continuation methods are defined to ensure that the record ends in SBCS mode.

Proper continuation of a statement from the end of an input record to the beginning of the next record depends on three factors:

- The number of bytes remaining for that record
- The character that precedes those remaining bytes
- The characters that are to fit into those bytes

The following table describes the conditions and methods of continuation. The rule numbers apply to Figure 1 on page 18.

Table 11. DBCS continuation rules

Rule number	Bytes remaining	Preceding character is	To be followed by	Remaining byte(s) should be	Next record begins
1	1	DBCS character or shift-out	DBCS character	Shift-in	Shift-out and DBCS character
2	2	DBCS character or shift-out	DBCS character	Shift-in and SBCS blank	Shift-out and DBCS character
3	1	DBCS character or shift-out	Shift-in and SBCS character	Shift-in	SBCS character
4	2	DBCS character or shift-out	Shift-in and SBCS character	Shift-in and SBCS character	
5	1	DBCS character or shift-out	Shift-in and SBCS blank	Shift-in	Required SBCS blank
6	2	DBCS character or shift-out	Shift-in and SBCS blank	Shift-in and continuing SBCS blank	Required SBCS blank
7	1	SBCS character	Shift-out	Not allowed. To avoid this case, rewrite or shift the record.	
8	2	SBCS character	Shift-out and DBCS character	Shift-out and shift-in (null string)	Shift-out and DBCS character
9	3	SBCS character	Shift-out and DBCS character	Shift-out and shift-in (null string) and continuing SBCS blank	Shift-out and DBCS character

Figure 1 on page 18 is a list of examples based on the rules shown in the above table. The examples use an assignment statement with a DBCS variable name and mixed character string. The examples show how continuation rules are maintained if the statement is split across two records (the vertical bars represent the left and right margins).

```
                              |←right margin
          B<kkkk>C='<kk> T'M;|
Rule                          |
No.                           |←left margin of next record
------------------------------------------------------------------------
             B<kkkk>C='<kk> T|
na                           |'M;
------------------------------------------------------------------------
              B<kkkk>C='<kk> |
6                            | T'M;
------------------------------------------------------------------------
               B<kkkk>C='<kk>|
5                            | T'M;
------------------------------------------------------------------------
                B<kkkk>C='<> |
2,9                          |<kk> T'M;
------------------------------------------------------------------------
                 B<kkkk>C='<>|
1,8                          |<kk> T'M;
------------------------------------------------------------------------
                  B<kkkk>C='<|
7                            | ...not allowed
------------------------------------------------------------------------
                   B<kkkk>C='|
na                           |<kk> T'M;
------------------------------------------------------------------------
                    B<kkkk>C=|
na                           |'<kk> T'M;
------------------------------------------------------------------------
                     B<kkkk>C|
4                            |='<kk> T'M;
------------------------------------------------------------------------
                      B<kkkk>|
3                            |C='<kk> T'M;
------------------------------------------------------------------------
                       B<kk> |
2                            |<kk>C='<kk> T'M;
------------------------------------------------------------------------
                        B<kk>|
1                            |<kk>C='<kk> T'M;
------------------------------------------------------------------------
                         B<> |
2,9                          |<kkkk>C='<kk> T'M;
------------------------------------------------------------------------
                          B<>|
1,8                          |<kkkk>C='<kk> T'M;
------------------------------------------------------------------------
                           B<|
7                            | ...not allowed
------------------------------------------------------------------------
                            B|
na                           |<kkkk>C='<kk> T'M;
```

Figure 1. Example of EBCDIC DBCS continuation rules

Chapter 2. Data elements

Chapter 2. Data elements

This chapter discusses the data you can use in a PL/I program:

- The types of data that are allowed
- The attributes that data can have
- The organization of the data

Variables and constants

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). Data items can be single items, called *scalars*, or they can be collections of items called *data aggregates*.

A *variable* has a value or values that can 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 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 3, "Expressions and references" on page 64.)

A *constant* has a value that cannot change. Constants for problem data are referred to by stating the value of the constant. Constants for program control data are referred to by name.

Data types

Data used in a PL/I program is either:

Problem data The coded arithmetic and string data types that a program processes.

Program control data The data types that control program execution: area, entry, event, file, label, offset, and pointer.

In the following example:

```
AREA = RADIUS**2*3.1416;
```

AREA and RADIUS are problem data variables. The numbers 2 and 3.1416 are problem data constants. The value of RADIUS is a data item, and the result of the computation can be either a data item to be assigned as the value of AREA, or a condition (OVERFLOW, for example, if the number is too long).

If the number 3.1416 is to be used in more than one place in the program, you can represent it as a variable that is assigned the value 3.1416. In this case, the above statement can be written as:

```
PI = 3.1416;
AREA = RADIUS**2*PI;
```

In the last statement, only the number 2 is a constant.

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

```
       GET LIST(B);
LOOP:  A=2*B;
       C=B+6;
```

Data attributes

The attributes of a variable or a program control data constant are not apparent in the name. Since PL/I must know these attributes, you can use keywords and expressions to specify the attributes explicitly in a DECLARE statement. Attributes can also be set contextually or by programmer-defined or language-specified defaults. For information on how the set of attributes is completed, see "Defaults for data attributes" on page 159.

Problem data constants also have attributes. The constant 3.1416 is coded arithmetic data type; the data item is a decimal number of five digits, and four of these digits are to the right of the decimal point. The constant 1.0 is different from the constants 1, '1'B, '1', 1B, or 1E0.

See Table 12 on page 23 and Table 13 on page 23 for the following:

- The classification of attributes according to data type.
- The valid combinations of attributes that can be applied to a data item.

  For a variable, attributes must be selected from the columns headed *Data attributes*, *Scope attributes*, *Storage attributes*, and *Alignment attributes*. For a constant, attributes must be selected from the columns *Data attributes* and *Scope attributes*.
- Language-specified default attributes. These are underlined.

In the following example:

```
DECLARE ST BIT(10);
```

by using Table 12 and Table 13, you can see that the default attributes AUTOMATIC, INTERNAL, and UNALIGNED apply to the name ST.

Table 12. Classification of variable attributes according to variable types

Variable type	Data attributes	Alignment attributes	Scope attributes	Storage attributes
Coded arithmetic[1]	REAL|COMPLEX FLOAT|FIXED BINARY|DECIMAL (precision)	ALIGNED| UNALIGNED	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) [INITIAL [CALL]][2,3,4] Defined variable: DEFINED [POSITION][5] Simple parameter: Parameter [CONNECTED] Controlled parameter: Parameter CONTROLLED [INITIAL]
String	BIT|CHARACTER|GRAPHIC (length) [VARYING]	ALIGNED| UNALIGNED		
Picture	PICTURE REAL|COMPLEX			
Label	LABEL	ALIGNED| UNALIGNED		
File	FILE[VARIABLE]			
Entry	ENTRY [RETURNS] [OPTIONS] [VARIABLE]			
Locator	POINTER| {OFFSET [(area-variable)]}			
Area	AREA(size)	ALIGNED		
Event	EVENT			

Aggregate variables:

Arrays: (dimension) may be added to the declaration of any variable.

Structures:

- For a major structure; scope, storage (except INITIAL), alignment, and the LIKE attribute may be specified.
- For a minor structure; alignment and the LIKE attribute may be specified.
- Members of structures have the INTERNAL attribute.

Notes:

1. Undeclared names, or names declared without a data type, default to coded arithmetic variables. Default attributes are described in "Defaults for data attributes" on page 159.
2. Arrays of nonstatic label variables may be initialized by subscripted label prefixes.
3. STATIC ENTRY and STATIC LABEL conflict with INITIAL (see INITIAL for a special case).
4. INITIAL CALL conflicts with STATIC.
5. POSITION can be used only with string overlay defining.

Table 13 (Page 1 of 2). Classification of constant attributes according to constant types

Constant type	Data attributes	Scope attributes
Coded arithmetic[1]	REAL|imaginary FLOAT|FIXED BINARY|DECIMAL (precision)	INTERNAL
String[1]	BIT|CHARACTER|GRAPHIC (length)	INTERNAL
Label[1]	LABEL	INTERNAL
File[2]	FILE ENVIRONMENT STREAM|RECORD INPUT|OUTPUT|UPDATE SEQUENTIAL|DIRECT BUFFERED|UNBUFFERED KEYED BACKWARDS PRINT	INTERNAL|EXTERNAL

Table 13 (Page 2 of 2). Classification of constant attributes according to constant types

Constant type	Data attributes	Scope attributes
Entry	ENTRY [RETURNS] [OPTIONS]	INTERNAL[1]|EXTERNAL
Built-in entry[3]	BUILTIN	INTERNAL

Notes:

1. Arithmetic constants, string constants, label constants, and INTERNAL entry constants cannot be declared in a DECLARE statement.
2. File attributes are described in "FILE attribute" on page 238.
3. Only mathematical built-in functions have entry values.

Problem data

There are two types of problem data: coded arithmetic and string. Figure 2 shows the types of problem data.

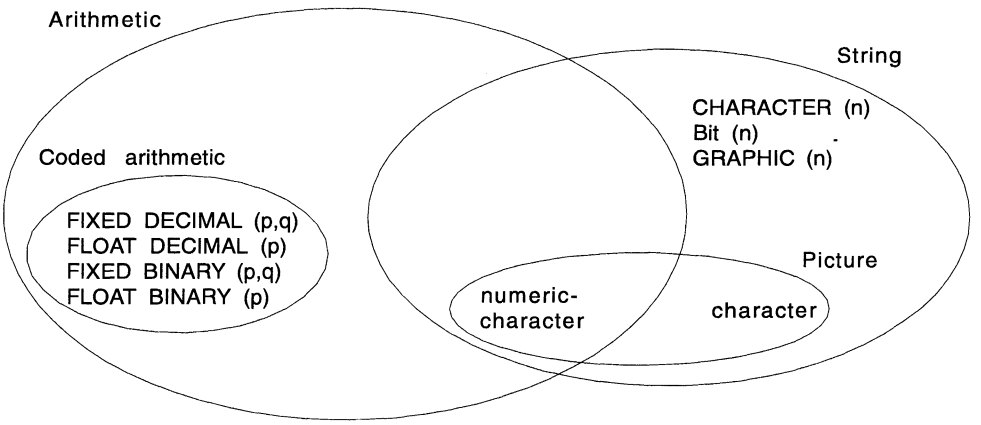


Figure 2. Types of problem data

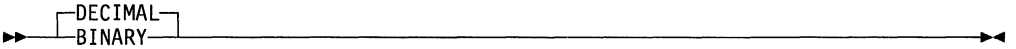
Arithmetic data is either coded arithmetic data or numeric character data. Numeric character data is a string data type; see "Numeric character data" on page 36 for more information. Coded arithmetic data items have the data attributes of base, scale, precision, and mode. Arithmetic values are rational numbers.

Coded arithmetic data and attributes

Coded arithmetic data attributes and coded arithmetic data types—decimal fixed-point, binary fixed-point, decimal floating-point, and binary floating-point—are described below.

BINARY and DECIMAL attributes

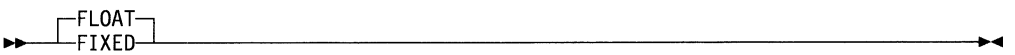
The *base* of a coded arithmetic data item is either decimal or binary. The syntax for the BINARY and DECIMAL attributes is:



Abbreviations: BIN for BINARY
DEC for DECIMAL

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. The syntax for the FIXED and FLOAT attributes is:

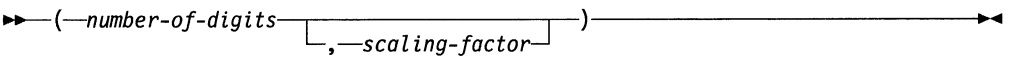


Precision attribute

The *precision* of a coded arithmetic data item is:

- For fixed-point, the number of digits the data item contains
- For floating-point, the number of significant digits to be maintained (excluding the exponent)

For fixed-point data items, the precision attribute can also specify the *scaling factor* (the assumed position of the decimal or binary point), relative to the rightmost digit of the number. The syntax for the precision attribute is:



number-of-digits
specifies an integer.

scaling-factor
(fixed-point only) specifies an optionally-signed integer. 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.

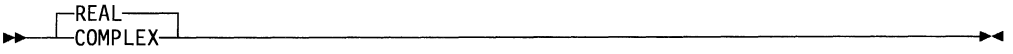
The precision attribute must follow, with no intervening attribute specifications, the scale (FIXED or FLOAT), base (DECIMAL or BINARY), or mode (REAL or COMPLEX) at the same factoring level.

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

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. The syntax for the REAL and COMPLEX attributes is:



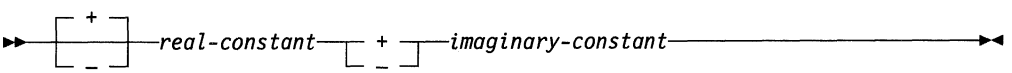
Abbreviation: CPLX for COMPLEX

Arithmetic variables default to REAL. 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 the examples above has a real part of zero. A complex value with a nonzero real part is represented by an expression with the 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 by:

```
REAL(x) = a*c - b*d
IMAG(x) = b*c + a*d
```

Computational conditions can be raised during these calculations.

Decimal fixed-point data

A decimal fixed-point constant consists of one or more decimal digits with an optional decimal point. If no decimal point appears, the point is assumed to be immediately to the right of the rightmost digit. 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)

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

```
DECLARE A FIXED DECIMAL (5,4);
```

specifies that A represents decimal fixed-point items of at least 5 digits, in the range -9.9999 to 9.9999.

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

DECLARE B FIXED DECIMAL(7);
```

both specify that B represents integers of at least 7 digits.

```
DECLARE C FIXED (7,-2) DECIMAL;
```

specifies that C has a scaling factor of -2. This means that C holds at least 7 digits in the range -9999999*100 to 9999999*100.

```
DECLARE D DECIMAL FIXED REAL(3,2);
```

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

The default precision is (5,0). Decimal fixed-point data is stored two digits to the 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 operations 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.

Binary fixed-point data

A binary fixed-point constant consists of one or more binary digits 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 binary digits in the constant, and *q* is the number of binary digits specified to the right of the binary point. Examples are:

Constant	Precision
10110B	(5,0)
11111B	(5,0)
101B	(3,0)
1011.111B	(7,3)

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

```
DECLARE FACTOR BINARY FIXED (20,2);
```

FACTOR is declared a variable that can represent binary fixed-point data items of 20 digits, two of which are fractional.

The default precision is (15,0). A binary fixed-point data item with:

- 8 to 15 digits is stored as a fixed-point binary halfword
- More than 15 digits, up to 31, is stored as a fullword

A halfword is 15 bits plus a sign bit; a fullword is 31 bits plus a sign bit.

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

Decimal floating-point data

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 followed by an optionally-signed integer, which specifies a power of ten. Decimal floating-point constants have a precision (p), where *p* is the number of digits of the mantissa. Examples are:

Constant	Precision
15E-23	(2)
15E23	(2)
4E-3	(1)
1.96E+07	(3)
438E0	(3)
3141593E-6	(7)
.003141593E3	(9)

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

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

```
DECLARE LIGHT_YEARS DECIMAL FLOAT(5);
```

LIGHT_YEARS represents decimal floating-point data items with at least 5 decimal digits.

The default precision is (6). Decimal floating-point data is stored as normalized hexadecimal floating-point, with the hexadecimal point assumed to the left of the first hexadecimal digit. 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.

Notes for extended precision floating-point data:

1. Avoid coding the internal representation of extended precision floating-point values. Under some circumstances, these values do not compare properly with other extended precision floating-point values. To avoid this problem, use decimal floating-point constants to specify extended precision floating-point values.
2. Extended precision divide operations are not supported on systems running in 370 mode (that is, VSE/ESA running on a 370-mode machine, or VSE/ESA

running as a 370-mode guest under VM). This restriction affects PL/I divide operations, the DIVIDE built-in function, and some of the LE/VSE mathematical services.

Binary floating-point data

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, followed by an optionally-signed decimal integer, which specifies a power of two. Binary floating-point constants have a precision (p) where *p* is the number of binary digits of the mantissa. Examples are:

Constant	Precision
101101E5B	(6)
101.101E2B	(6)
11101E-28B	(5)
11.01E+42B	(4)

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 items with a precision of 16 binary digits.

The default precision is (21). The exponent cannot exceed three decimal digits. Binary floating-point data is stored as normalized hexadecimal floating-point. 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.

Notes for extended precision floating-point data:

1. Avoid coding the internal representation of extended precision floating-point values. Under some circumstances, these values do not compare properly with other extended precision floating-point values. To avoid this problem, use binary floating-point constants to specify extended precision floating-point values.
2. Extended precision divide operations are not supported on systems running in 370 mode (that is, VSE/ESA running on a 370-mode machine, or VSE/ESA running as a 370-mode guest under VM). This restriction affects PL/I divide operations, the DIVIDE built-in function, and some of the LE/VSE mathematical services.

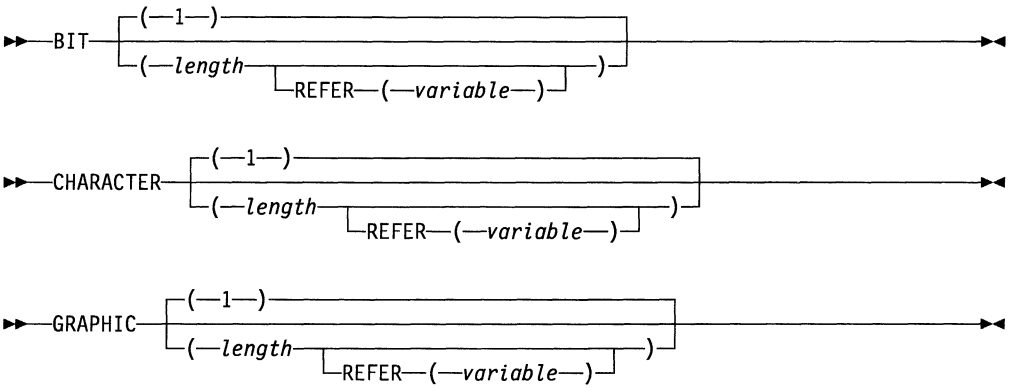
String data and attributes

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

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:



Abbreviations: CHAR for CHARACTER
G for GRAPHIC

length Specifies the length of a fixed-length string or the maximum length of a varying-length string. If length is not specified, the default is 1. The length is in bits, characters, or graphics (DBCS characters), as appropriate.

You can specify length by an expression or an asterisk, but certain restrictions apply when specifying the length specifications of the elements of data aggregates in parameter descriptors. Expressions can be used only for controlled parameters, and asterisks must not be used if a dummy is created in the corresponding argument.

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.

The asterisk notation can be used for parameters or controlled variables. The length can be taken from a previous allocation or, for CONTROLLED variables, it can be specified in a subsequent ALLOCATE statement.

If a string has the STATIC attribute, length must be an integer.

If a string has the BASED attribute, length must be an integer unless the string is a member of a based structure and the REFER option is used (see "REFER option (self-defining data)" on page 215).

REFER See "REFER option (self-defining data)" on page 215 for a description of the REFER option.

The statement below declares USER as a variable that can represent character data items with a maximum length of 15:

```
DECLARE USER CHARACTER (15);
```

Character variables can also be declared using the PICTURE attribute.

The following example shows the declaration of a bit variable:

```
DECLARE SYMPTOMS BIT (64);
```

VARYING attribute

VARYING specifies that the variable is to represent varying-length strings, in which case length (in the BIT, CHARACTER, or GRAPHIC attribute) specifies the maximum length. The syntax for the VARYING attribute is:

Abbreviation: VAR

The length at any time is the length of the current value. The storage allocated for varying-length strings is 2 bytes longer than the declared maximum length. The leftmost 2 bytes hold the string's current length (in bytes for a character variable, bits for a bit variable, or graphics for a graphic variable).

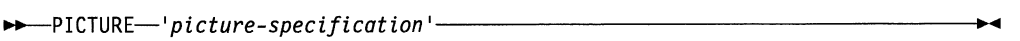
The following DECLARE statement specifies that USER represents varying-length character data items with a maximum length of 15:

```
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 length at any given time by use of the LENGTH built-in function.

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 the characters that can occupy that position. The syntax for the PICTURE attribute is:



Abbreviation: PIC

picture-specification
is either character-picture-specification or numeric-picture-specification and is discussed in detail in Chapter 13, "Picture specification characters" on page 304.

A 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.

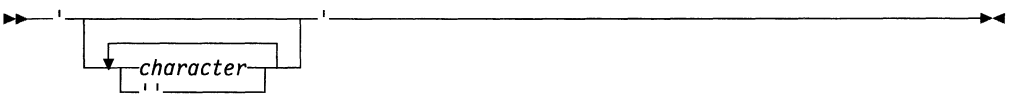
For more information on numeric character data, see "Numeric character data" on page 36.

Character data

When declared with the CHARACTER attribute, a character value can include any digit, letter, special character, blank, or any other of the 256 EBCDIC codes. When declared with the PICTURE attribute, the character value assigned must match the picture-specification. Each character of a character value occupies 1 byte of storage.

Character constant

A character constant is a contiguous sequence of characters enclosed in single quotation marks. If a single quotation mark is a character in a string, it must be written as two single quotation marks with no intervening blank. The length of a character constant is the number of characters between the enclosing quotation marks. However, if two *single* quotation marks are used within the string to represent a single quotation mark, they are counted as a single character. The syntax for a character constant is:



A null character constant is written as two quotation marks with no intervening blank.

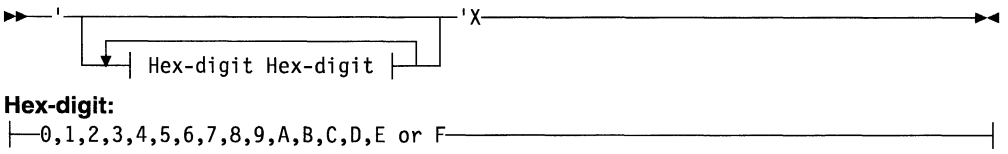
Examples of character constants are:

Constant	Length
'LOGARITHM TABLE'	15
'PAGE 5'	6
'SHAKESPEARE''S ''HAMLET'''	22
'AC438-19'	8
'/* THIS IS NOT A COMMENT */'	27
''	0
(2)'WALLA '	12

In the last example, the parenthesized number is a *string repetition factor*, which indicates repetition of the characters that follow. This example specifies `'WALLA WALLA '`. The string repetition factor must be an integer, enclosed in parentheses.

X (Hexadecimal) character constant

The X string constant describes a character string constant in hexadecimal (hex) notation. You can use the X constant to manipulate nonprintable character strings. You can also use the X string constant in preprocessor statements. The syntax for a hex character constant is:



Hex-digit:

The data type remains CHARACTER, and the padding/truncation rules are the same as character string constants.

You must specify an even number of hexadecimal digits.

Examples of X character string constants are:

```
'D7D361C9'X     is the same as  'PL/I'
(2)'C1'X        is the same as  (2)'A'
(2)'C1'X        is the same as  'AA'
'7D'X           is the same as  ''''
''X             is the same as  ''
```

Note: Using the X constant may make the program not portable to other implementations. For example, '4E'X is the same as '+' on EBCDIC machines, but is treated as 'N' on ASCII machines.

Bit data

When declared with the BIT attribute, a bit value can contain bits 0 and 1. A collection of 8 or less unaligned bits occupy 1 byte of storage.

Bit constant

A bit constant is represented by a series of the digits 0 and 1, enclosed in single 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 with no intervening blank, followed immediately by the letter B.

Examples of bit constants are:

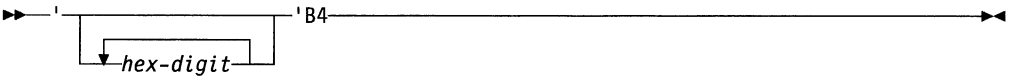
Constant	Length
'1'B	1
'11111010110001'B	14
(64)'0'B	64
''B	0

The parenthesized number in the third constant is a string repetition factor which specifies that the following series of digits is to be repeated the specified number of times. The example shown would result in a string of 64 binary zeros.

(See "Source-to-target rules" on page 93 for a discussion on the conversion of BIT-to-CHARACTER data and CHARACTER-to-BIT data.)

B4 (Bit Hexadecimal) bit hexadecimal constant

The B4 string constant describes a bit string constant in hexadecimal notation. You can use it any place that you use a bit string constant. The B4 constant is a convenient way to represent long bit strings. The syntax for a bit constant is:



The data type remains BIT and padding/truncation rules are the same as for bit string constants.

Note: The term BX is a synonym for B4.

Some examples of B4 string constants are:

```
'CA'B4     is the same as  '11001010'B
'80'B4     is the same as  '10000000'B
'1'B4      is the same as  '0001'B
(2)'F'B4   is the same as  '11111111'B
(2)'F'B4   is the same as  (2)'1111'B
''B4       is the same as  ''B
```

Graphic data

When declared with the GRAPHIC attribute, a graphic value can contain any graphic, each occupying 2 bytes of storage.

Graphic constant

A graphic constant can contain values in the range X'00' to X'FF' in both bytes. However, you cannot use X'0E' and X'0F'. The constant must contain an even number of bytes, including zero (a null graphic constant). The syntax for a graphic constant is:

The GRAPHIC compile-time option must be in effect for graphic constants to be accepted. The GRAPHIC ENVIRONMENT option must be specified for STREAM I/O files having graphic constants; if not, the conversion condition is raised.

Enclose the graphic values in quotes and use the G suffix after them. The quotes can be either single byte (outside the shift-out shift-in pair, <>) or double byte (inside the shift-out shift-in pair). However, the beginning and ending quote must match. The G suffix can also be single or double byte.

The preferred form for a graphic constant is:

```
'<gggg>'G
```

where gg is a valid two-byte graphic.

Other allowable forms are:

```
<.'gggg.'>G
<.'gggg.'.G>
```

Some of the ways to represent the null graphic constant are:

```
'<>'G
<.'.'>G
<.'.'.G>
''G
```

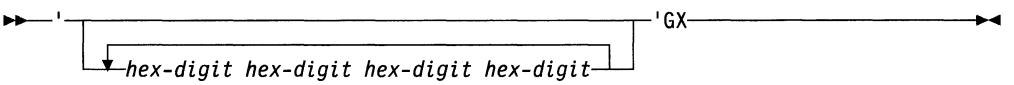
If you include a DBCS quotation mark within a graphic constant, and DBCS quotes enclose the graphic constant, you must double the included quote. The following examples show graphic constants that contain one DBCS quote:

```
<.'.'.'.'>G          /* contained quote doubled */
'<.'>'G              /* single contained quote  */
```

Graphic strings require 2 bytes of storage for each DBCS character.

GX (Graphic Hexadecimal) graphic string constant

The GX string constant describes a GRAPHIC constant using hexadecimal notation. The data type remains GRAPHIC and follows the rules for GRAPHIC data. You can use GX string constants wherever you use GRAPHIC string constants. You do not need the GRAPHIC compile-time option or the GRAPHIC ENVIRONMENT option to use the GX string constant. The syntax for a GX string constant is:



Four hexadecimal digits (two bytes) represent each graphic element. A GX string constant must contain a multiple of four digits. Enclose the digits in quotes, followed by the GX suffix.

Example:

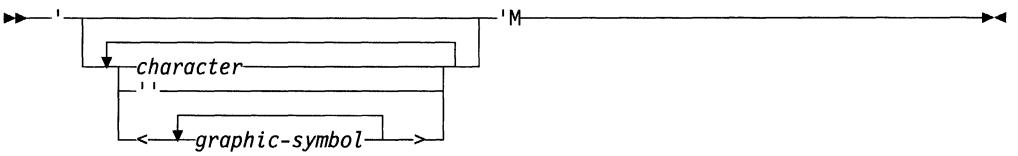
```
'42C142C242C3'GX      is the same as  '<.A.B.C>'G
''GX                  is the same as  ''G
```

Mixed data

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

M (Mixed) string constant

The GRAPHIC compile-time option must be in effect for mixed constants to be accepted. The GRAPHIC ENVIRONMENT option must be specified for STREAM I/O files having mixed constants; if not, the CONVERSION condition is raised. The syntax for a mixed string constant is:



Quotes must enclose a mixed character string constant (either SBCS or DBCS), and the M suffix must follow it. The data always begins and ends in SBCS, and shift codes enclose the DBCS portions.

The following rules apply:

- Shift-out/shift-in pairs must match; you cannot nest pairs.
- The DBCS portion cannot contain '0E'X or '0F'X in either byte.
- The character portion cannot contain the values '0E'X or '0F'X, unless specifically intended as shift codes.

The preferred form for a mixed character string is:

```
'cc<kkkk>'M
```

Other allowable forms are:

```
<.'kkkk>cc<.'.M>
<.'>cc<kkkk.'.M>
```

where cc is CHARACTER data, kk is a DBCS character

Examples:

'IBM'M	is the same as	'IBM'
'<.I.B.M>'M	is stored as	<.I.B.M>
<.'kkkk.'>.M	is adjusted and stored as	<kkkk> (6 bytes)
''M	is the same as	''

Numeric character data

A numeric character data item is the value of a variable that has been declared with the PICTURE attribute and 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 more information on numeric picture specifications, see Chapter 13, "Picture specification characters" on page 304. For example:

```
DECLARE PRICE PICTURE '999V99';
```

specifies that any value assigned to PRICE is maintained as a character string of 5 decimal digits, with an assumed decimal point preceding the rightmost 2 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 automatic, but require extra execution 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 a 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 the same. 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 arithmetic 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 CONVERSION 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, since 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 complete discussion of picture characters, see Chapter 13, “Picture specification characters.”

PLIXOPT variable

If you want to override the default run-time options that were established during installation, you can use the PLIXOPT variable. PLIXOPT is an external static varying character string containing the list of options you want to use. The options can be separated by either blanks or commas. For information about run-time options, see the *LE/VSE Programming Guide*.

Program control data

The types of program control data are: area, entry, event, file, label, pointer, and offset.

Area data and attribute

For a description of area data and the AREA attribute, see "Area data and attribute" on page 219.

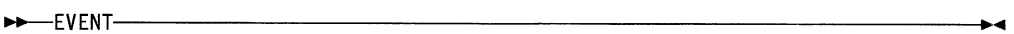
Entry data and attribute

For a description of entry data and the ENTRY attribute, see "Entry data" on page 133.

Event data and attribute

Event variables are used to allow a degree of overlap between a record-oriented input/output operation (or the execution of a DISPLAY statement) and the execution of other statements in the procedure that initiated the operation.

A variable is given the EVENT attribute by its appearance in an EVENT option or a WAIT statement, or by explicit declaration. The syntax for the EVENT attribute is:



An event variable has two separate values:

- A single bit reflecting the completion value of the variable. '1'B indicates complete, and '0'B indicates incomplete.
- A real fixed-point binary value of precision (15,0) reflecting the status value of the variable. A zero value indicates normal status; a nonzero value indicates abnormal status.

The values of the event variable can be set by one of the following means:

- Using the COMPLETION pseudovariable, to set the completion value
- Using the STATUS pseudovariable, to set the status value
- By an event variable assignment
- By a statement with the EVENT option
- By a WAIT statement for an event variable associated with an input/output event or DISPLAY statement
- By closing a file on which an input/output operation with an EVENT option is in progress

On allocation of an event variable, its completion value is '0'B (incomplete). The status value is undefined.

To associate an event variable with an event (an input/output operation or DISPLAY), you use the EVENT option of a statement. The event variable remains associated with the event until the event is completed. For an input/output (or DISPLAY) event, the event is completed during the execution of the WAIT for the associated event which initiated the operation. During this period the event variable is active.

It is an error to:

- Associate an active event variable with another event
- Modify the completion value of an active event variable by event variable assignment or by use of the COMPLETION pseudovariable
- Assign a value to an active event variable (including an event variable in an array, structure, or area) by means of an input/output statement

The values of the event variable can be returned separately by use of the COMPLETION and STATUS built-in functions.

Assignment of one event variable to another assigns both the completion and status values.

For further information, see "EVENT option" on page 259 and "DISPLAY statement" on page 174.

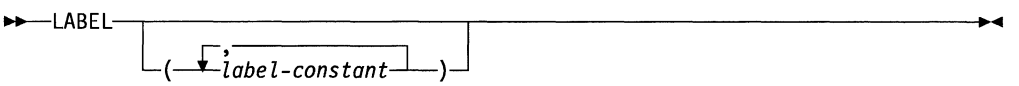
File data and attribute

For a description of file data and the FILE attribute, see "FILE attribute" on page 238.

Label data and attribute

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

The LABEL attribute specifies that the name being declared is a label variable and has label-constants as values. To aid in optimization of the object program, the attribute specification can also include the values that the name can have during execution of the program. The syntax for the LABEL attribute is:



If a list of label-constants is given, the variable must have as its value a member of the list when used in a GO TO statement or R-format item. The label-constants in the list must be known in the block containing the declaration. The maximum allowable number of label-constants in the list is 125.

The parenthesized list of label-constants can be used in a LABEL attribute specification for a label array.

A label constant is 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.

```
ABCDE: MILES = SPEED*HOURS;
```

In this example, ABCDE is a label constant. The statement can be executed either by normal sequential execution of instructions or by transferring control to this statement from some other point in the program by means of a GO TO statement.

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.

An alternate method of initialization (the INITIAL attribute can also be used) is available for elements of arrays of non-STATIC label variables. An initial value of an element can appear as a statement prefix, provided that its subscript is an optionally-signed integer. This initializes that array element to a value that is a label constant for the prefixed statement. The prefixed statement must be internal to the block containing the declaration of the array. Only one form of initialization can be used for a given label array.

In the following example, transfer is made to a particular element of the array Z by giving I a value of 1, 2, or 3. If I=2, omitting Z(2) would cause an error.

```
DECLARE Z(3) LABEL;
GO TO Z(I);
  .
  .
  .
Z(1):  IF X = Y THEN RETURN;
  .
  .
  .
Z(2):  A = A + B + C * D;
  .
  .
  .
Z(3):  A = A + 10;
```


Offset data and attribute

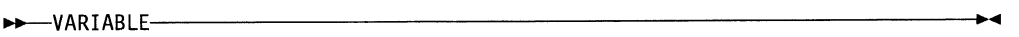
Offset data is one of two types of locator data. For a description of offset data and the OFFSET attribute, see "Offset data and attribute" on page 220.

Pointer data and attribute

Pointer data is one of two types of locator data. For a description of pointer data and the POINTER attribute, see "POINTER variable and attribute" on page 212.

VARIABLE attribute

The VARIABLE attribute can be specified only with the ENTRY or FILE attributes. It establishes the name as an entry variable or a file variable. The syntax for the VARIABLE attribute is:



The VARIABLE attribute is implied if the name is an element of an array or structure, or if any of the following attributes is specified:

- Storage class attribute
- Parameter
- Alignment attribute
- DEFINED
- INITIAL

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 shown in Table 14.

Table 14. 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							

It is possible in PL/I to align data on integral boundaries. This is not always desirable, however, since there may be unused bytes between successive data elements, which increases use of storage. This increase is particularly important 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

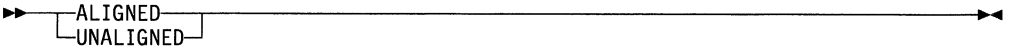
ALIGNED specifies that the data element is aligned on the storage boundary corresponding to its data-type requirement. These requirements are shown in Table 15.

Table 15. Alignment requirements

Variable type	Stored internally as:	Storage requirements (in 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: 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)		
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) 1<=p<=15	Halfword	2		
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	–	8		
LABEL	–			
EVENT	–	32		EVENT and AREA data cannot be unaligned
AREA	–	16+size	Doubleword (Data may begin on byte 0)	
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<=109	Extended floating-point	16		
DECIMAL FLOAT (p) 17<=p<=33				

This table shows 4-byte locators.
The alignment requirements for 16-byte locators are identical to those for 4-byte locators.
Storage requirements are system dependent. The storage requirements listed above are for the 370 system. Storage values can be determined using the STORAGE or CURRENTSTORAGE built-in functions.

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:



Although the UNALIGNED attribute can reduce storage requirements, it can also increase run time.

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.

For all operators and user-defined and built-in functions, the default for ALIGNED or UNALIGNED is applicable to the elements of the result.

Constants take the default for ALIGNED or UNALIGNED.

ALIGNED or UNALIGNED can be specified for element, array, or structure variables. The application of either attribute to a structure 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    */
```

Data aggregates

Data items can be single data elements, or they can be grouped together to form *data aggregates*. Data aggregates are groups of data items that can be referred to either collectively or individually and can be either arrays or structures.

A variable that represents a single element is an *element variable* (also called a scalar variable). A variable that represents an aggregate of data elements is either an *array variable* or a *structure variable*.

Any type of problem data or program control data can be grouped into arrays or structures. (The examples of arrays in this chapter show arrays of arithmetic data).

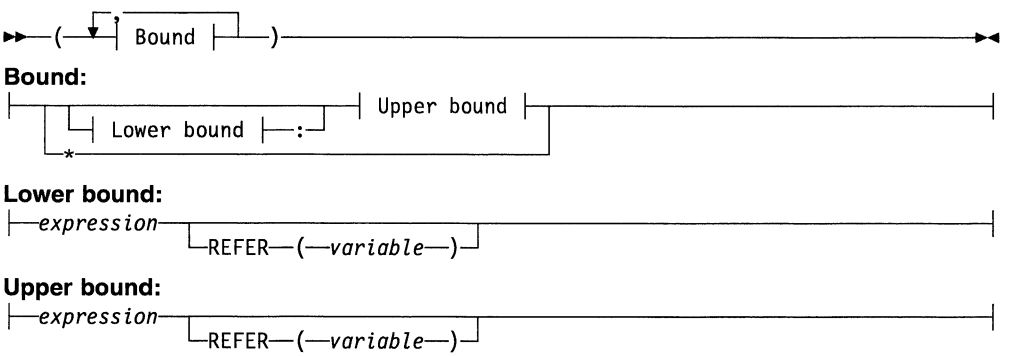
Arrays

An *array* is an n-dimensional collection of elements that have identical attributes. Only the array itself is given a name. You refer to an individual item of an array by giving its position within the array.

The parenthesized number or numbers following the array name in a DECLARE statement is the dimension attribute specification.

Dimension attribute

The *dimension attribute* specifies the number of dimensions of an array. This attribute also specifies either the bounds of each dimension or indicates that the upper bound is taken from an associated argument. The *bounds* of a dimension are the beginning and the end of that dimension. The *extent* is the number of integers between, and including, the lower and upper bounds. The syntax for the dimension attribute is:



The number of bounds specifications indicates the number of dimensions in the array, unless the declared variable is in an array of structures. In this case it inherits dimensions from the containing structure.

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 actual bounds are specified in an ALLOCATE statement, if the variable is CONTROLLED, or in a declaration of an associated argument, if the variable is a simple parameter.

When allocating storage for an array, PL/I converts *bounds* that are expressions to FIXED BINARY (31,0). The bounds of arrays declared STATIC must be optionally-signed integers. The bounds of arrays declared BASED must be optionally-signed integers unless the array is part of a based structure and the REFER option is used (see "REFER option (self-defining data)" on page 215).

The dimension attribute must follow, with no intervening attribute specifications, the array name (or the parenthesized list of names, if it is being factored).

Variables that are associated with arrays should be declared FIXED BINARY (31,0). These include:

- Variables used for indexing
- Variables used to contain array bounds
- Variables and expressions that receive or manipulate values that array-related built-in functions (such as HBOUND, LBOUND, and DIM) return

Examples of arrays

Consider the following two declarations:

```
DECLARE LIST (8) FIXED DECIMAL (3);
```

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; its extent is 8.

```
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; the extents are 4 and 2.

Other examples are:

```
DECLARE LIST_A (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; in each case, there are 8 integers from the lower bound through the upper bound.

When manipulating the array data of more than one array (discussed in "Array expressions" on page 82), 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. 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, that is, with the right-most subscript varying 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 *unconnected* storage (see "CONNECTED attribute" on page 230). The rule is as follows: if a nonasterisk bound appears to the right of the leftmost asterisk bound, the array cross section is in unconnected storage. Thus A(4,*,*) is in connected storage; A(*,2,*) is not.

Structures

A *structure* is a data aggregate whose elements need not have identical attributes. Like an array, the entire structure is given a name that can be used to refer to the entire aggregate of data. Unlike an array, however, each element of a structure also has a name.

A structure has different *levels*. At the first level is the structure name called a *major structure*. At a deeper level are the names of substructures called *minor structures*. At the deepest are the element names called *elementary* names. An elementary name in a structure can represent an array, in which case it is not an element variable, but an array variable.

The organization of a structure is specified 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 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 can be declared as follows:

```
DECLARE 1 PAYROLL,
          2 NAME,
            3 LAST CHAR(20),
            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);
```

Indentation is only for readability. The statement could be written in a continuous string as DECLARE 1 PAYROLL, 2 NAME, 3 LAST CHAR(20), etc.

PAYROLL is declared as a major structure containing the minor structures NAME, HOURS, and RATE. Each minor structure contains two elementary names. 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 an element by referring to an elementary name.

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*. PAYROLL might have been declared as follows:

```
DECLARE 1 PAYROLL,
          4 NAME,
            5 LAST CHAR(20),
            5 FIRST CHAR(15),
          2 HOURS,
            6 REGULAR FIXED DEC(5,2),
            5 OVERTIME FIXED DEC(5,2),
          2 RATE,
            3 REGULAR FIXED DEC(3,2),
            3 OVERTIME FIXED DEC(3,2);
```

This declaration results in exactly the same structuring as the previous declaration.

The description of a major structure name is terminated by one of the following:

- The declaration of another item with a level number 1
- The declaration of another item with no level number
- A semicolon terminating the DECLARE statement

Structure-qualification

If there is no ambiguity, you can refer to a minor structure or a structure element by the minor structure name or the elementary name alone.

A *qualified reference* is an elementary name or a minor structure name that is qualified with one or more names at a higher level, connected by periods. Blanks may appear surrounding the period.

Structure-qualification is in the order of levels; that is, the name at the highest level must appear first, with the name at the deepest level appearing last.

Names within a structure, except the major structure name itself, need not be unique within the procedure in which it is declared. Qualifying names must be used only so far as necessary to make the reference unique. In the previous example, the qualified reference PAYROLL.LAST is equivalent to the name PAYROLL.NAME.LAST.

An *ambiguous reference* is a reference with insufficient qualification to make the reference unique. 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:

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

In this example, A.C refers to C in the inner block; D.E refers to E in the outer block.

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

In this example, B has been declared twice. A reference to A.D refers to the second D, since A.D is a complete qualification of only the second D; the first D would have to be referred to as A.C.D.

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

In this example, a reference to A.C is ambiguous because neither C can be completely qualified by this reference.

```
DECLARE 1 A, 2 A, 3 A;
```

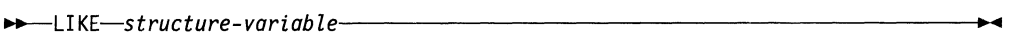
In this example, a reference to A refers to the first A, A.A to the second A, and A.A.A to the third A.

```
DECLARE X;
DECLARE 1 Y, 2 X, 3 Z, 3 A,
             2 Y, 3 Z, 3 A;
```

In this 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.

LIKE attribute

The LIKE attribute specifies that the variable being declared is a structure variable with the same structuring as the variable following the attribute keyword LIKE. Substructure names, elementary names, and their attributes are identical. The syntax for the LIKE attribute is:



structure-variable

can be a major structure or a minor structure. It can be qualified, but it cannot be subscripted. It must not contain a REFER variable.

The *structure-variable* must be known in the block containing the LIKE attribute specification. The structure names in all LIKE attributes are associated with declared structures before any LIKE attributes are expanded.

Neither the *structure-variable* nor any of its substructures can be declared with the LIKE attribute.

Structure-variable must not be a substructure of a structure declared with the LIKE attribute.

No substructure of the major structure containing *structure-variable* can have the LIKE attribute.

Additional substructures or elementary names cannot be added to the created structure; any level number that immediately follows the *structure-variable* in the LIKE attribute specification in a DECLARE statement must be equal to or less than the level number of the name declared with the LIKE attribute.

Attributes of the *structure-variable* itself do not carry over to the created structure. For example, storage class attributes do not carry over. If the *structure-variable* following the keyword LIKE represents an array of structures, its dimension attribute is not carried over. However, attributes of substructure names and elementary names are carried over; contained dimension and length attributes are recomputed.

If a direct application of the description to the structure declared LIKE causes an incorrect continuity of level numbers (for example, if a minor structure at level 3 were declared LIKE a major structure at level 1) the level numbers are modified by a constant before application.

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 *structure-variable*. The only ALIGNED and UNALIGNED attributes that are carried over from the *structure-variable* are those explicitly specified for its substructures and elements. For example:

```
DECLARE 1 A, 2 C, 3 E, 3 F,
        1 D(10), 2 C, 3 G, 3 H;
   .
   .
   .
BEGIN;
DECLARE 1 A LIKE D;
DECLARE 1 B(2) LIKE A.C;
   .
   .
   .
END;
```

These declarations result in the following:

```
1 A LIKE D is expanded to give:

    1 A, 2 C, 3 G, 3 H

1 B(2) LIKE A.C is expanded to give:

    1 B(2), 3 E, 3 F
```

The following declaration is **invalid**:

```
DECLARE 1 A BASED,
          2 X FIXED BINARY,
          2 Y(Z REFER(X)),
        1 B BASED LIKE A;
```

because references to the REFER object X would be ambiguous.

The following example is **invalid**:

```
DECLARE 1 A LIKE C,
        1 B,
          2 C,
            3 D,
            3 E LIKE X,
          2 F,
        1 X,
          2 Y,
          2 Z;
```

because the LIKE attribute of A specifies a structure, C, that contains an identifier, E, that has the LIKE attribute.

The following example is **invalid**:

```
DECLARE 1 A LIKE G.C,
        1 B,
          2 C,
            3 D,
            3 E,
          2 F,
        1 G LIKE B;
```

because the LIKE attribute of A specifies a substructure, G.C, of a structure, G, declared with the LIKE attribute.

The following example is **invalid**:

```
DECLARE 1 A LIKE C,
        1 B,
          2 C,
            3 D,
            3 E,
          2 F LIKE X,
        1 X,
          2 Y,
          2 Z;
```

because the LIKE attribute of A specifies a structure, C, within a structure, B, that contains a substructure, F, having the LIKE attribute.

Arrays of structures

A structure name, either major or minor, can be given a dimension attribute in a DECLARE statement to declare an *array of structures*—an array whose elements are structures having identical names, levels, and elements. For example, if a structure, WEATHER, is used to process meteorological information for each month of a year, it might be declared as follows:

```
DECLARE 1 WEATHER(12),
          2 TEMPERATURE,
            3 HIGH DECIMAL FIXED(4,1),
            3 LOW DECIMAL FIXED(3,1),
          2 WIND_VELOCITY,
            3 HIGH DECIMAL FIXED(3),
            3 LOW DECIMAL FIXED(2),
          2 PRECIPITATION,
            3 TOTAL DECIMAL FIXED(3,1),
            3 AVERAGE DECIMAL FIXED(3,1);
```

Thus, you could refer to the weather data for the month of July by specifying WEATHER(7). Portions of the July weather could be referred to by TEMPERATURE(7) and WIND_VELOCITY(7). PRECIPITATION(7) or TOTAL(7) would both refer to the total precipitation during the month of July.

TEMPERATURE.HIGH(3), which would refer to the high temperature in March, is a *subscripted qualified reference*.

The need for subscripted qualified references becomes apparent when an array of structures contains minor structures that are arrays. For example, consider the following array of structures:

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

Both A and B are arrays of structures. 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 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 the right or left and attached to names at a lower or higher level. For 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*; thus, A.B(1,1,2).C has interleaved subscripts.

Any item declared within an array of structures inherits dimensions declared in the containing structure. For example, in the above 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

A reference to a cross section of an array of structures is not allowed—that is, the asterisk notation cannot be used in a reference.

Structure mapping

For any major or minor structure, the length, alignment requirement, and position relative to a doubleword 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.

During the structure mapping process, the compiler 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 following rule to each item in turn, starting at the beginning of the structure declaration:

The logical level of a given item is always one unit deeper than that of its immediate containing structure.

For example:

```
DCL 1 A, 4 B, 5 C, 5 D, 3 E, 8 F, 7 G;

    1    2    3    3    2    3    3
```

The lower line shows the logical level for each item in the declaration.

Rules for order of pairing

To determine the order of pairing:

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" on page 55).
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 the purpose of this explanation, think of storage as contiguous doublewords, each having 8 bytes, numbered 0 through 7, indicating 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 depends 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.

When this process is complete, any padding between the two elements has been minimized and will not change for 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.

EVENT and AREA data cannot be unaligned. If a structure has the UNALIGNED attribute and contains an element that cannot be unaligned, UNALIGNED is ignored for that element; the compiler aligns the element and produces an error. For example, in a program with the declaration:

```
DECLARE 1 A UNALIGNED,
          2 B,
          2 C AREA(100);
```

C is given the attribute ALIGNED, because the inherited attribute UNALIGNED conflicts with AREA.

Example of structure mapping

This example shows the application of the structure mapping rules for a structure declared as follows:

```
DECLARE 1 A ALIGNED,
          2 B FIXED BIN(31),
          2 C,
            3 D FLOAT DECIMAL(14),
            3 E,
              4 F LABEL,
              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(5),
              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.

For each minor structure, the table in Figure 3 shows the steps in the process; Figure 4 shows a graphic representation of the process. Finally, the major structure A is mapped as shown in Figure 5. At the end of the example, the structure map for A is set out in the form of a table (see Figure 6) showing the offset of each member from the start of A.

	Name of element	Alignment requirement	Length	Offset from doubleword		Length of padding	Offset from minor structure
				Begin	End		
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 3. Mapping of example structure

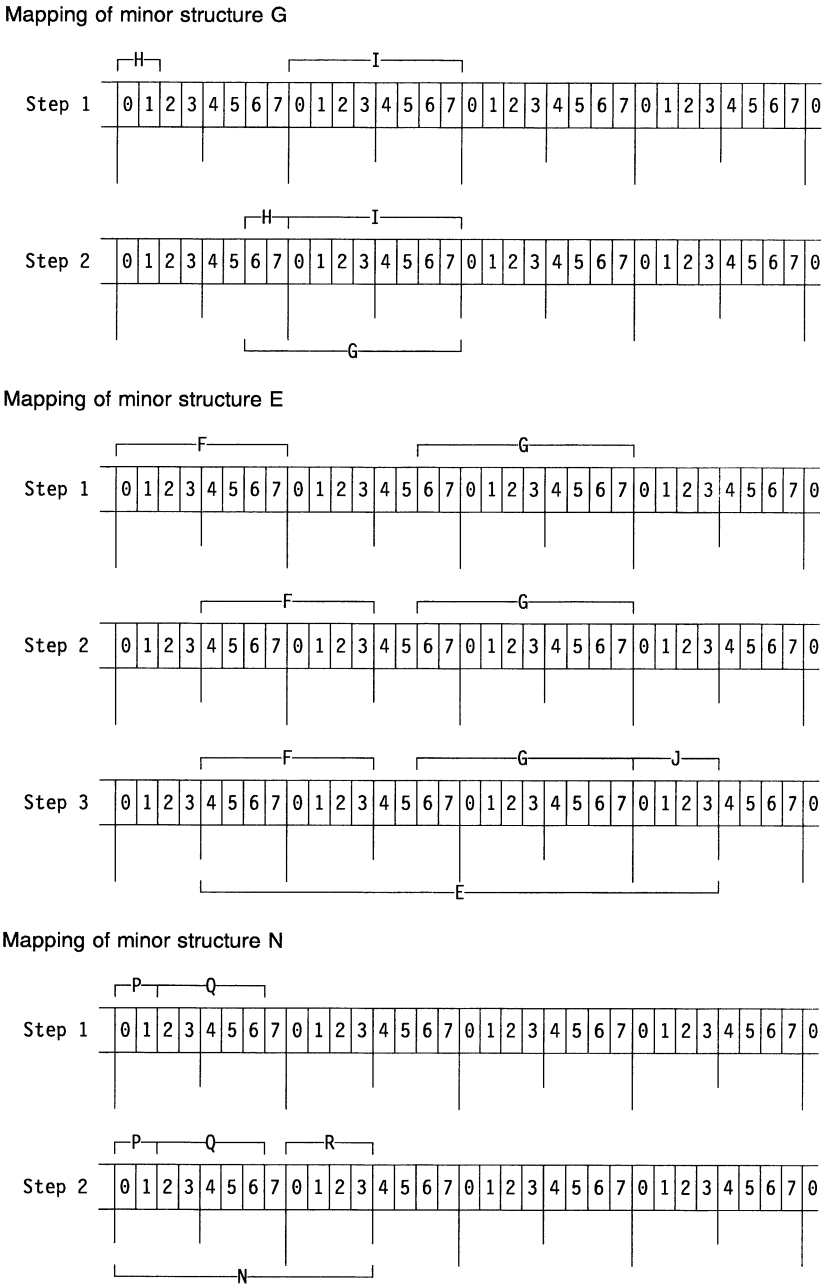
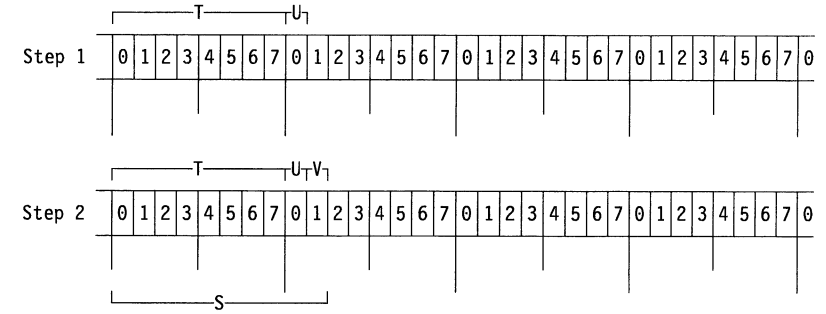
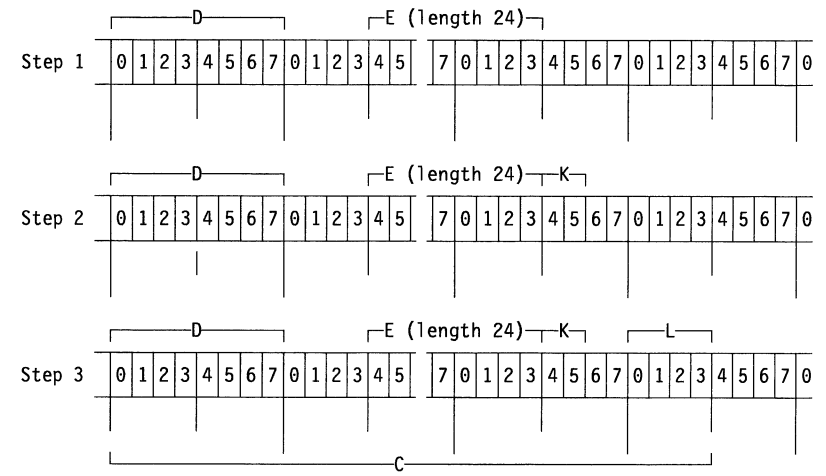


Figure 4 (Part 1 of 2). Mapping of minor structures

Mapping of minor structure S



Mapping of minor structure C



Mapping of minor structure M

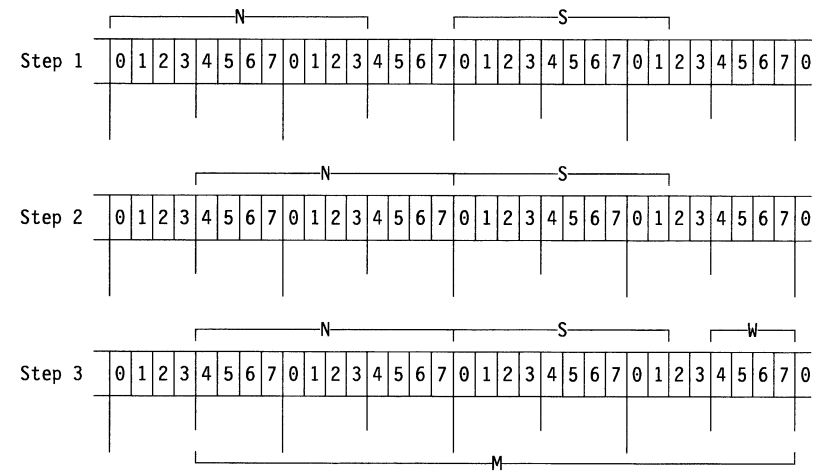


Figure 4 (Part 2 of 2). Mapping of minor structures

	Name of item	Alignment required	length	Offset from doubleword		Length of padding	Offset from A
				Begin	End		
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

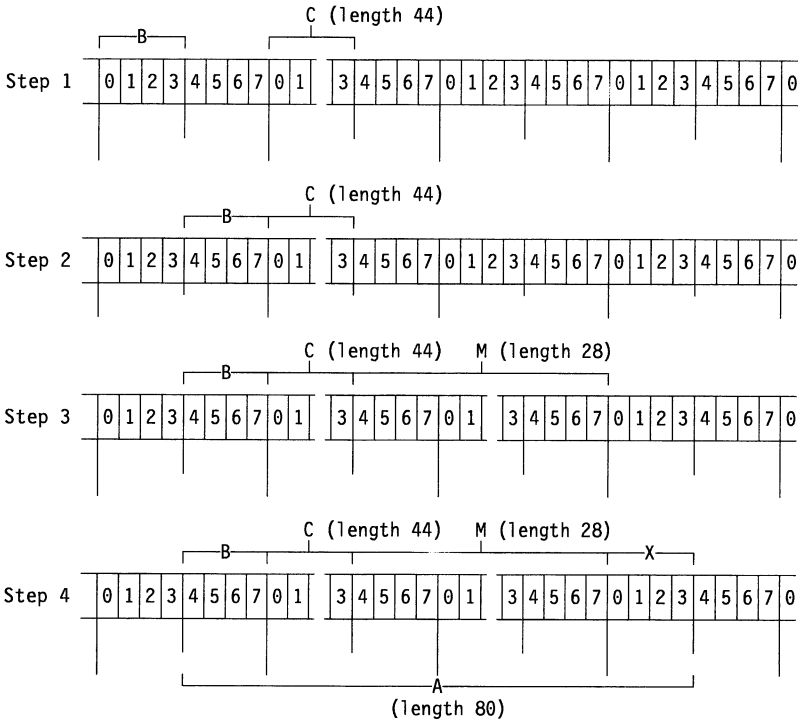


Figure 5. Mapping of major structure A

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 6. Offsets in final mapping of structure A

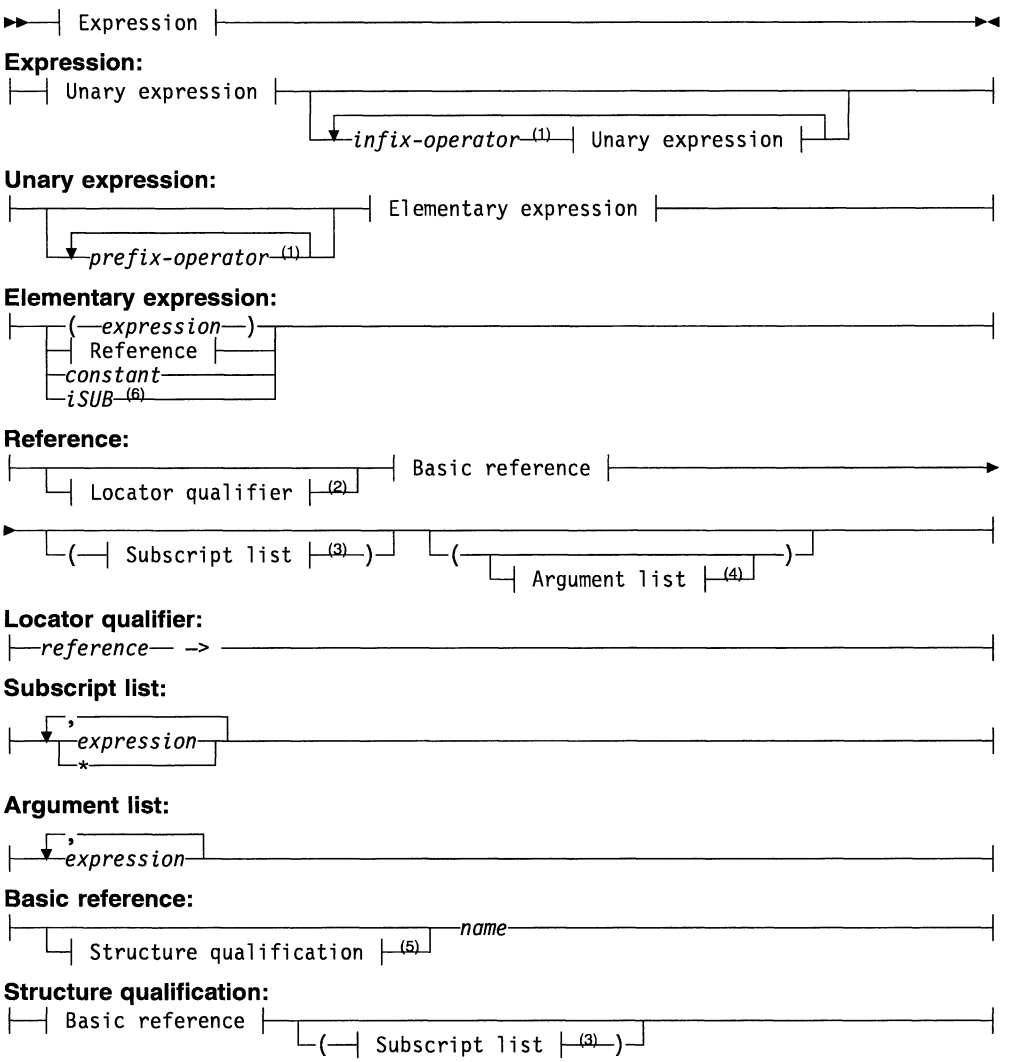
Chapter 3. Expressions and references

Chapter 3. Expressions and references

This chapter discusses the various types of expressions and references.

An expression is a representation of a value. A single constant or a variable or a function reference is an expression. Combinations of constants and/or variables and/or function references, along with operators and/or parentheses, are expressions. An expression that contains operators is an *operational expression*. The constants and variables of an operational expression are called *operands*.

The syntax for expressions and references is shown in the following diagram.



Notes:

1 Operators are shown in Table 10 on page 8.

2 Locator-qualifier is described under "Locator qualification" on page 211.

3 Subscripts are described under "Arrays" on page 45.

4 Arguments are described in "Association of arguments and parameters" on page 128.

5 Structure-qualification is described under "Structures" on page 48.

6 iSUBs are described in "DEFINED attribute" on page 224.

Any expression can be classified as an *element expression* (also called a scalar expression), an *array expression*, or a *structure expression*. Element variables, array variables, and structure 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 subscripted name that specifies a single element of an array.

An array expression represents an array of values. This definition includes a structure, or part of a structure (a minor structure or element) that has the dimension attribute.

A structure expression represents a structured set of values. None of its operands are arrays, but an operand can be subscripted.

In the examples that follow, the variables have attributes declared as follows:

```
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);
```

Examples of element expressions are:

```
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)
```

Examples of array expressions are:

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

The last example represents an array of structures.

Examples of structure expression are:

```
RATE * COST(2)
RATE / 2
```

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 or a structure 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, given the following function:

```
INC: PROC(P) RETURNS (FIXED BIN(31));
     DCL P FIXED BIN(31);
     P = P + 1;
     RETURN (P);
     END INC;
```

then, in the following, the initial values of the elements of A could be 1,2 or 2,1 or even 1,1:

```
I = 0;
    BEGIN;
    DCL A(2) FIXED BIN(31) INIT((2)(INC(I)));
    END;
```

Similarly, for the following, the array element referenced could be B(1,2) or B(2,1) or B(1,1):

```
DCL B(2,2);
I = 0;
PUT LIST (B(INC(I),INC(I)));
```

You should not assume that the evaluation order of an expression is necessarily left to right. For example, in the IF expression below, the function reference CHK might be invoked before the FLG bit variable is tested.

```
DCL FLG BIT;
DCL CHK ENTRY RETURNS(BIT);
IF FLG & CHK() THEN ...
```

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 20th 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.

For more information, see "Pseudovariables" on page 363.

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 4, "Data conversion" on page 88.

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 four classes of operation: arithmetic, bit, comparison, and concatenation.

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, 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 Table 16 on page 71. Each operand is converted to the attributes of the result.)

Type: Character operands are converted to FIXED DECIMAL (15,0). Bit operands are converted to FIXED BINARY (31,0). Numeric character operands are converted to DECIMAL with scale and precision determined by the picture-specification. 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 this 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 this 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:

- If the exponent precision is other than (p,0), both operands are converted to floating-point.
- If the exponent is not an unsigned integer, the first operand is converted to floating-point.
- if precisions indicate that the result of the fixed-point exponentiation exceeds the maximum number of digits allowed (15 decimal digits or 31 binary digits), the first operand is converted to floating-point.

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.

Table 16 shows the attributes and precisions that result from various arithmetic operations, and Table 17 shows the attributes of the result for the special cases of exponentiation noted in the right-hand column of Table 16.

Table 16. Results of arithmetic operations for coded arithmetic operands

1st operand	2nd operand	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 = 15 q = 15-p_1+q_1-q_2	FLOAT DECIMAL (p) (unless special case A applies) p = MAX(p_1,p_2)
FIXED BINARY (p_1,q_1)	FIXED BINARY (p_2,q_2)	FIXED BINARY (p,q)			p = 31 q = 31-p_1+q_1-q_2	FLOAT BINARY (p) (unless special case B applies) p = MAX(p_1,p_2)
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)	FIXED 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)					
FIXED DECIMAL (p_1,q_1)	FIXED BINARY (p_2,q_2)	FIXED BINARY (p,q)	p = 1+MAX(r-s,p_2-q_2)+q q = MAX(s,q_2)	p = 1+r+p_2 q = s+q_2	p = 31 q = 31-r+s-q_2	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)	FLOAT BINARY (p_2)	FLOAT BINARY (p)	p = MAX(CEIL(p_1*3.32),p_2)			
FLOAT DECIMAL (p_1)	FIXED BINARY (p_2,q_2)					
FLOAT DECIMAL (p_1)	FLOAT BINARY (p_2)					
FIXED BINARY (p_1,q_1)	FIXED DECIMAL (p_2,q_2)	FIXED BINARY (p,q)	p = 1+MAX(p_1-q_1,t-u)+q q = MAX(q_1,u)	p = 1+p_1+t q = q_1+u	p = 31 q = 31-p_1+q_1-u	FLOAT BINARY (p) (unless special case B or C applies) p = MAX(p_1,CEIL(p_2*3.32))
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_1)	FIXED DECIMAL (p_2,q_2)					
FLOAT BINARY (p_1)	FLOAT DECIMAL (p_2)					

where:

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)

The calculations of precision values cannot exceed the implementation maximum:

FIXED BINARY:	31	FLOAT BINARY:	109
FIXED DECIMAL:	15	FLOAT DECIMAL:	33

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

Notes:

1. Special cases of exponentiation are described in Table 17 on page 72.
2. For a table of CEIL(N*3.32) values, see Table 20 on page 92.

Table 17. Special cases for exponentiation

Case	First operand	Second operand	Attributes of result
A	FIXED DECIMAL (p_1,q_1)	Integer with value n	FIXED DECIMAL (p,q) (provided p <= 15) where p = (p_1 + 1)*n-1 and q = q_1*n
B	FIXED BINARY (p_1,q_1)	Integer with value n	FIXED BINARY (p,q) (provided p <= 31) where p = (p_1 + 1)*n-1 and q = q_1*n
C	FLOAT (p_1)	FIXED (p_2,0)	FLOAT (p_1) 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.

The intermediate result has attributes, like any variable in a PL/I program. The attributes of the intermediate result depend on the attributes of the two operands (or, in the case of a prefix operation, the single operand) and on the operator involved. This dependence is further explained under "Targets" on page 67.

If arithmetic is performed on character data, the intermediate results are held in the maximum fixed decimal precision (15,0). In the following example:

```
DCL A CHAR(6) INIT('123.45');
DCL B FIXED(5,2);
B=A;    /* B HAS VALUE 123.45 */
B=A+A;  /* B HAS VALUE 246.00 */
```

The ADD, MULTIPLY, and DIVIDE built-in functions in PL/I 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
```

would be undefined and FIXEDOVERFLOW would be raised. To obtain the result 25.3333333333333, write

```
25+01/3
```

The explanation is that constants have the precision with which they are written, while FIXED division results in a value of maximum implementation defined precision. The results of the two evaluations are reached as follows:

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)
01	(2,0)	01
e	(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:

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

Bit operations

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

```
¬    &    |
```

The first operator, the *not* symbol, can be used as a prefix operator only. The second and third operators, the *and* symbol and the *or* symbol, can be used as infix operators only. (The operators have the same function as in Boolean algebra.)

Note: The NOT and OR compile-time options can specify alternative characters to be used as not and or symbols in your program source. See the *PL/I VSE Programming Guide* for details.

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. The following table shows the result for each bit position for each of the operators:

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

In the following examples:

The value of operand A is '010111'B
The value of operand B is '111111'B
The value of operand C is '110'B
The value of operand D is 5

```
¬ A                 yields  '101000'B
¬ C                 yields  '001'B
C & B               yields  '110000'B
A | B               yields  '111111'B
C | B               yields  '111111'B
A | (¬C)            yields  '011111'B
¬((¬C)|(¬B))        yields  '110111'B
SUBSTR(A,1,1)|(D=5) yields  '1'B
```

BOOL built-in function

In addition to the *NOT*, *AND*, and *OR* operations using the operators ¬, &, and |, Boolean operations can be performed using the BOOL built-in function.

Comparison operations

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

```
<   ¬<   <=   =   ¬=   >=   >   ¬>
```

These operators specify *less than*, *not less than*, *less than or equal to*, *equal to*, *not equal to*, *greater than or equal to*, *greater than*, and *not greater than*, respectively.

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.

Comparisons can be:

Algebraic

Compares signed arithmetic values in coded arithmetic form. If operands differ in base, scale, precision, or mode, they are converted as described under "Data conversion in arithmetic operations" on page 69. Before comparison, numeric character data is converted to coded arithmetic. Only the operators = and ¬= are valid for comparison of complex operands.

Character
Compares characters according to the binary value of their bytes, left-to-right, character-by-character.

Bit Compares binary digits left-to-right, bit-by-bit.

Graphic
Compares the binary values of DBCS characters left-to-right, symbol-by-symbol.

Pointer data
Compares pointer values containing any relational operators. However, the only conversion that can take place is offset to pointer, and only if the comparison is = or ¬=. Under the compile-time option LANGLVL(SPROG), pointer expressions containing any of the 8 infix operators can be compared.

Program control data
Compares the internal coded forms of the operands. Only the comparison operators = and ¬= are allowed; area variables cannot be compared. All other type differences between operands for program control data comparisons are in error.

Comparisons are equal for the following operands under these conditions:

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.

Entry names on the same PROCEDURE or ENTRY statement do not compare equal.

Event If both the status and completion values are 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.

Offset If values are equal.

If the operands of a problem 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) character, (3) 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
padding graphic 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, the value '1'B is assigned to X if A is less than B; otherwise, the value '0'B is assigned:

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

In the following assignment statement:

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

the first = symbol is the assignment symbol; the second = 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.

Tables for comparison operations

Figure 7 and Table 18 show the attributes to which the two operands of a comparison operation are converted before they are compared. The figures show the type of comparison that is made. To use these tables:

1. Refer first to Figure 7.
2. Find the entry in Figure 7 that corresponds to the two operands in the expression you are evaluating. The entry is two numbers separated by a comma.
3. Refer to Table 18 with the entry numbers for your expression. The numbers indicate the attributes to which each operand is converted. The first number gives the attributes to which the first operand is converted, and the second number those to which the second operand is converted.

The following example shows how to use Figure 7 and Table 18:

```
DECLARE ITEM CHARACTER(5),
    STANDARD FIXED BINARY(15,0);
IF ITEM¬=STANDARD THEN DO;
```

1. In the example, the first operand has the CHARACTER attribute and the second operand has the FIXED BINARY attribute. The corresponding entry in Figure 7 has the numbers 13 and 1.
2. Entry 13 in Table 18 shows attributes FIXED BINARY (31,0). This entry indicates that ITEM is converted to coded arithmetic form with these attributes. Entry 1 in Table 18 is *No conversion,* indicating that STANDARD is not converted. Both entries show that the comparison will be algebraic. (The two entries in Table 18 will always show the same type of comparison.)

3. The tables indicate that ITEM is converted to FIXED BINARY (31,0). ITEM is then compared algebraically with STANDARD, whose attributes remain FIXED BINARY (15,0).

Maximum precisions for arithmetic data: Table 18 gives formulas for calculating precision. The actual precision values can never exceed the maximum number of digits allowed (as listed in the Appendix, "PL/I limits" on page 434).

				Second operand							
				Coded arithmetic				Numeric character (PICTURE)		CHARACTER(n2)	BIT(n2)
				FIXED		FLOAT					
First operand				DECIMAL (p2,q2)	BINARY (p2,q2)	DECIMAL (p2,q2)	BINARY (p2,q2)	Fixed point	Floating point		
	Coded arithmetic	FIXED	DECIMAL (p1,q1)	1,1	4,1	5,1	8,1	1,10	5,11	1,12	4,13
			BINARY (p1,q1)	1,4	1,1	7,6	7,1	1,4	7,6	1,13	1,13
		FLOAT	DECIMAL (p1)	1,5	6,7	1,1	6,1	1,5	1,11	1,14	6,9
			BINARY (p1)	1,8	1,7	1,6	1,1	1,8	1,6	1,15	1,9
	Numeric character (PICTURE)	Fixed point		10,1	4,1	5,1	8,1	10,10	5,11	10,12	4,13
		Floating point		11,5	6,7	11,1	6,1	11,5	11,11	11,14	6,9
	CHARACTER(n1)			12,1	13,1	14,1	15,1	12,10	14,11	2,2	2,2
	BIT(n1)			13,4	13,1	9,6	9,1	13,4	9,6	2,2	3,3

Figure 7. Index table for comparison operations

Note: If one operand is COMPLEX and the other is REAL, the REAL operand is converted to COMPLEX before the comparison is made.

Table 18 (Page 1 of 2). Comparison operations

Code	Type of comparison	Attributes of comparison operand
1	Algebraic	No conversion
2	Character	CHARACTER (MAX(n_1,n_2)) where (n_1) and (n_2) are the lengths of the first and second operands, respectively.
3	Bit	BIT (MAX(n_1,n_2)) where (n_1) and (n_2) are the lengths of the first and second operands, respectively.
4	Algebraic	FIXED BINARY (1+CEIL(p*3.32),CEIL(ABS(q*3.32))*SIGN(q)) where (p,q) is the precision of operand being converted. (If operand is in numeric character (PICTURE) form, see note below).
5	Algebraic	FLOAT DECIMAL (p) where (p) is the precision of operand being converted. (If operand is in numeric character (PICTURE) form, see note below).
6	Algebraic	FLOAT BINARY (CEIL(p*3.32)) where (p) is the precision of operand being converted. (If operand is in numeric character (PICTURE) form, see note below).
7	Algebraic	FLOAT BINARY (p) where (p,q) is the precision of operand being converted. (If operand is in numeric character (PICTURE) form, see note below).

If the operand being converted is in numeric character form, its precision is implied by the PICTURE specification.

Table 18 (Page 2 of 2). Comparison operations

Code	Type of comparison	Attributes of comparison operand
8	Algebraic	FLOAT BINARY (CEIL(p*3.32)) where (p) is the precision of operand being converted. (If operand is in numeric character (PICTURE) form, see note below).
9	Algebraic	FLOAT BINARY (31)
10	Algebraic	FIXED DECIMAL (Precision same as implied by PICTURE specification of operand being converted)
11	Algebraic	FLOAT DECIMAL (Precision same as implied by PICTURE specification of operand being converted)
12	Algebraic	FIXED DECIMAL (15,0)
13	Algebraic	FIXED BINARY (31,0)
14	Algebraic	FLOAT DECIMAL (15)
15	Algebraic	FLOAT BINARY (50)
16	Algebraic	FLOAT DECIMAL (10)

If the operand being converted is in numeric character form, its precision is implied by the PICTURE specification.

Concatenation operations

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

||

Note: The OR compile-time option can specify alternate characters to be used as the OR symbol. These characters can then be combined to form the concatenation 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 string type since concatenation can be performed only upon strings—either character, bit, or graphic. If either operand is graphic, both must be graphic. If either operand is character or decimal, any necessary conversions are performed to produce a character result. Otherwise, the operands are bit or binary, and conversions are performed to produce a bit result.

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 example:

```
The value of operand A is '010111'B
The value of operand B is '101'B
The value of operand C is 'XY,Z'
The value of operand D is 'AA/BB'

A||B     yields  '010111101'B
A||A||B  yields  '010111010111101'B
C||D     yields  'XY,ZAA/BB'
```

```
D||C     yields  'AA/BBXY,Z'
B||D     yields  '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 consisting of 8 characters.

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 "and" operation is performed.
- The result of the "and" 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. Such is the case for this particular expression. The order of evaluation, however, depends upon the priority of the operators appearing in the expression.

Priority of operators

Table 19 shows the priority of operators in expression evaluations.

Table 19 (Page 1 of 2). 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
	¬	Bit string	Operand is converted to FIXED BINARY if it is a BIT string
			All non-BIT data converted to BIT

Table 19 (Page 2 of 2). Priority of operations and guide to conversions

Priority	Operator	Type of operation	Remarks
2	*, /	Arithmetic	Result is in coded arithmetic form
3	infix +, –	Arithmetic	Result is in coded arithmetic form
4	||	Concatenation	If one or both operands are CHARACTER or DECIMAL, non-CHARACTER operand(s) converted to CHARACTER
			If operands are BIT and BINARY or both operands are BINARY, non-BIT operand(s) converted to BIT
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

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 (DIIE**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 *and* 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.

Pointer operations

The following pointer support extensions can be used when LANGLVL(SPROG) is in effect:

- Add an expression to or subtract an expression from a pointer expression. The expression type must be BIT, REAL FIXED BINARY(p,0), or REAL FIXED DECIMAL(p,0). If necessary, the nonpointer operand will be converted to REAL FIXED BINARY(31,0). 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 BINARY(31,0) 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 BIT, REAL FIXED BINARY(p,0), or REAL FIXED DECIMAL(p,0) 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 REAL FIXED BINARY(31,0).

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.

For a description of the LANGLVL compile-time option, see the *PL/I VSE Programming Guide.*

Array expressions

Evaluating 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.

PL/I array expressions are not generally the expressions of conventional matrix algebra.

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, another array, or a structure 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
```

The original value for A(1,2), which is 10, is used in the evaluation for only the first two elements of A. Since the result of the expression is assigned to A, changing the value of A, the new value of A(1,2) is used for all subsequent operations. The first two elements are multiplied by 10, the original value of A(1,2); all other elements are multiplied by 100, the new value of A(1,2).

Using operations that involve elements of the same array, as in the above example, often can produce unexpected results. It is recommended that you assign such an element value to a temporary variable and use that temporary variable in this array-and-element expression unless, of course, the desired results are produced.

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 bit strings of length 1

```
1  0  0
0  0  1
0  1  1
```

The following example tests that all the elements of an array are in ascending sequence:

```
DCL X(100),
    XA(99) DEFINED(X),
    XB(99) DEFINED(X(1SUB+1));

IF ALL(XA < XB)
   THEN GOTO ASCENDING;
```

Array-and-structure operations

If the two operands of an infix operator are an array and a structure, the result is an array of structures with bounds identical to the array. The structuring of the result is identical to the original structure. In the following example:

```
DECLARE 1 A, 2 B, 2 C,
        X(2),
        Y(2) LIKE A;

        Y = X + A;
```

is equivalent to:

```
Y.B(1) = X(1) + A.B;
Y.C(1) = X(1) + A.C;
Y.B(2) = X(2) + A.B;
Y.C(2) = X(2) + A.C;
```

If the structure has a dimension attribute on the level-1 name, the operation becomes an array-and-array operation. If the array elements are structures, the rules about identical structuring given under "Structure expressions" apply to the array elements and the structure.

Structure expressions

Element variables and constants can be operands of a structure expression. Evaluation of a structure expression yields a structure result. A structure operand can be a major structure name or a minor structure name. All operations performed on structures are performed element-by-element.

All structure variables appearing in a structure expression must have identical structuring, which means:

- The structures must have the same minor structuring and the same number of contained elements and arrays.
- The positioning of the elements and arrays within the structure (and within the minor structures, if any) must be the same.
- Arrays in corresponding positions must have identical bounds.

Names do not have to be the same. Data types of corresponding elements do not have to be the same, as long as valid conversion can be performed.

Prefix operators and structures

The result of the operation of a prefix operator on a structure is a structure of identical structuring, each element of which is the result of the operation having been performed upon each element of the original structure.

Since structures can contain elements of many different data types, a prefix operation in a structure expression would be meaningless unless the operation can be validly performed upon every element represented by the structure variable, which is either a major structure name or a minor structure name.

Infix operators and structures

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

Structure operands in a structure expression need not be major structure names. A minor structure name, at any level, is a structure variable. Thus, if N is a minor structure in the major structure M, the following is a structure expression:

```
M.N & '1010'B
```

Structure-and-element operations

When an infix operator has one structure and one element operand, it is the same as a series of operations, one for each element in the structure.

Consider the following structure:

```
1 A,
  2 B,
    3 C,
    3 D,
    3 E,
  2 F,
    3 G,
    3 H,
    3 I;
```

If X is an element variable, then A * X is equivalent to:

```
A.C * X
A.D * X
A.E * X
A.G * X
A.H * X
A.I * X
```

Structure-and-structure operations

When an infix operator has two structure operands, it is the same as a series of element operations, one for each corresponding pair of elements. For example, if A is the structure shown in the previous example and M is the following structure:

```
1 M,
  2 N,
    3 O,
    3 P,
    3 Q,
  2 R,
    3 S,
    3 T,
    3 U;
```

then A || M is equivalent to:

```
A.C || M.O
A.D || M.P
A.E || M.Q
A.G || M.S
A.H || M.T
A.I || M.U
```

Chapter 4. Data conversion

Chapter 4. Data conversion

This chapter discusses data conversions for problem 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 210.

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

The tables that follow summarize the circumstances that can cause conversion.

The following can cause conversion to any 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 or ENTRY	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 or ENTRY 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
WAIT	expression option
Most statements	subscript

All of the attributes, except string lengths, of both the source data item and the target data item must be known at compile time. Conversion can raise one of the following conditions (which are described in Chapter 15, "Conditions"): SIZE, CONVERSION, STRINGSIZE, or OVERFLOW.

There are no conversions of graphic data to or from other data types.

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, 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 problem data conversion

Conversions can take place during expression evaluation and on assignment. Conversions can also be initiated with built-in functions (see Chapter 16, "Built-in functions, subroutines, and pseudovariables").

The functions provided for conversion are:

```
BINARY      DECIMAL      GRAPHIC
BIT         FIXED        IMAG
CHAR        FLOAT        REAL
COMPLEX
```

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

The PRECISION built-in function controls the precision of data.

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, the target, or both 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 bits. 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

- Numeric character PICTURE
- CHARACTER
- BIT

Changes in value can occur in converting between decimal representations and binary representation (FLOAT DECIMAL has a 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 Table 20 to calculate these conversions.

*Table 20. Table of 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	54	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

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.

Numeric 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 (15,0) 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 is 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
The source bit string is converted to an unsigned binary value with precision of (15,0) if the conversion occurs during evaluation of an operational expression, or with precision of (56,0) if the conversion occurs during an assignment. The greater precision is possible in an assignment because the compiler can readily

determine the final target. 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.

Target: FIXED BINARY (p2,q2)

Source:

FIXED DECIMAL (p1,q1)
The precision of the result is p2=MIN(1+CEIL(p1*3.32),31) and q2=CEIL(ABS(q1*3.32))*SIGN(q1). If p1 exceeds 14, the OVERFLOW condition might be raised. If the calculated value of p2 exceeds 31, significant digits on the left might be lost. This raises the SIZE condition.

FLOAT BINARY (p1)
The precision conversion is as described under "Converting arithmetic precision" on page 91 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.

Numeric character PICTURE
CHARACTER
BIT
See Target: coded arithmetic on page 93.

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 91 with p1 as declared or indicated and q1 as indicated by the decimal point position and modified by the value of the exponent.

Numeric character PICTURE
CHARACTER
BIT
See Target: coded arithmetic on page 93.

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).

Numeric character PICTURE
CHARACTER
BIT
See Target: coded arithmetic on page 93.

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).

Numeric character PICTURE
CHARACTER
BIT
See Target: coded arithmetic on page 93.

Target: numeric character PICTURE

The numeric character PICTURE data item is the character representation of a decimal fixed-point or floating-point value. The following descriptions for source to

numeric 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.

Numeric 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.

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.

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{+|-}nnn
```

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'
```

Numeric character PICTURE
A real numeric character field is interpreted as a character string and assigned to the target string according to the rules for converting string lengths. If the numeric 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.

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))
```

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

Examples

Example of 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.

Example of 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 BINARY(4,0), taking 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 are converted 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.

Example of 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'*/
```

Example of 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 5. Program organization

Chapter 5. Program organization

This chapter describes how to organize statements into blocks to form a PL/I program, and how control flows within a program from one block of statements to another.

Proper division of a program into blocks simplifies the writing and testing of the program, particularly when a number of 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.

This chapter also describes subroutines and functions:

Statements for subroutines and functions
Association of arguments and parameters
Parameter attributes
Generic attributes and references
Passing an argument to the main procedure or another program
How to determine an entry value

Subroutines or functions can:

- Be invoked from different points in a program to perform the same frequently-used process
- Process data passed from different points of invocation
- Return control and, in the case of functions, return a value derived from the execution of the function to a point immediately following the point of invocation
- Be internal or external to the invoking block
- Have one or more entry points

Subroutines and functions can make use of data known in the invoking block by:

- Arguments and parameters: references in the invoking block are passed to the invoked procedure by an argument list associated with a CALL statement or option, or function reference. These references are made available by parameters in the invoked procedure.
- Using names whose scope of declaration includes both the invoking block and the invoked procedure. "Scopes of declarations" on page 154 describes the scope of declarations.

Programs

A PL/I program consists of one or more external procedures. Each procedure can contain other procedures, begin-blocks, or both.

The *LE/VSE Programming Guide* describes how external procedures can be either linked together or dynamically fetched.

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 (but not another PL/I program). The main procedure must be an external procedure whose PROCEDURE statement has the OPTIONS(MAIN) specification. For example:

```
CONTRL: PROCEDURE OPTIONS(MAIN);
        CALL A;
        CALL B;
        CALL C;
        END CONTRL;
```

In this example, CONTRL is the main procedure and it invokes other procedures in the program.

The main procedure remains active for the duration of the program.

Program termination

A program is terminated when the main procedure is terminated (see "Procedure termination" on page 119). On termination, whether normal or abnormal, control is returned to the calling program.

Blocks

A block is a delimited sequence of statements that determines the scope of the declaration of names declared within it, limits the allocation of automatic variables, and determines the scope of DEFAULT statements (as described in Chapter 6, "Data declaration" on page 150). A block is generally used to define a sequence of related operations to be performed on specified data.

There are two kinds of blocks: procedure blocks (or, simply, *procedures*) and begin blocks.

Begin blocks and procedures can contain declarations that are treated as local definitions of names. This is done to determine the scope of the declaration of 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 Chapter 6, "Data declaration" on page 150 for more information.

Automatic storage is allocated upon entry to the block where the storage is declared. The storage is freed upon exit from the block. See Chapter 8, "Storage control" on page 200 for more information.

Block activation

Although the begin block and the procedure play the same role in the allocation and freeing of storage and in delimiting the scope of names, they differ in the way they are activated and executed:

- Except for the main procedure, external and internal procedures contained in a program are activated only when they are invoked by a procedure reference.
- Begin blocks are activated through sequential flow or as ON-units.

During block activation:

- Expressions for automatic and defined variables are evaluated for dimension bounds, area sizes, string lengths, and initial values (including iteration factors).
- Storage is allocated for automatic variables and initialization, if specified.
- Currently active blocks known to the procedure are identified, so that the correct generations of automatic storage are accessible, and the correct ON-units may be entered.
- Storage is allocated for dummy arguments that might be created in this block.

The compiler assigns values in the following order for each block in the program.

1. Values that are independent of other declarations in the block. (Values may be inherited from an outer block).
2. Values that are dependent on other declarations in the block. If a value depends on more than one other declaration in the block, correct initialization is not guaranteed. In the following example, correct initialization of *K* is not guaranteed:

   ```
   DCL I INIT(10),J INIT(I),K INIT(J);
   ```

Declarations of data items must not be mutually dependent. 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 condition) 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. Particularly vulnerable to this situation are PUT DATA statements in ON-units established prior to block entry. They imply reference to automatic variables in all active blocks.

Block termination

A procedure is terminated when control passes back to the invoking block or to some other active block, by means other than a procedure reference. Similarly, a begin-block is terminated when control passes to another active block, by means other than a procedure reference. There are a number of ways that control can be transferred, and their interpretations differ according to the type of block being 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.

Internal and external blocks

Any block can contain one or more blocks; that is, procedures and begin-blocks can contain other procedures and begin-blocks. However, there can be no overlapping of blocks; a block that contains another block must totally encompass that block.

A procedure that is contained within another block is called an *internal procedure*. A procedure that is not contained within another block is called an *external procedure*. Each external procedure is compiled separately.

Begin-blocks are always internal; they must always be contained within another block.

Internal procedures and begin-blocks can be nested. Nested blocks, in turn, can have blocks nested within them, and so on. The outermost block must be a procedure. For example:

```
A:  PROCEDURE;
    .
    .
    B: BEGIN;
       .
       .
       END B;
    .
    .
    C: PROCEDURE;
       .
       .
       D: BEGIN;
          .
          .
          E: PROCEDURE;
             .
             .
             END E;
          .
          .
          END D;
       END C;
    .
    .
    END A;
```

Procedure A is an external procedure because it is not contained in any other block. Begin-block B is contained in A; it contains no other blocks. Internal procedure C contains begin-block D, which, in turn, contains internal procedure E. This example shows a depth of nesting of three levels relative to A; B and C are at a depth of one, D is at a depth of two, and E is at a depth of three.

Procedures

A procedure is a sequence of statements delimited by a PROCEDURE statement and a corresponding END statement. For example:

```
NAME: A:
   PROCEDURE;
    .
    .
    .
   END NAME;
```

The leftmost label of the PROCEDURE statement represents the *primary entry point* of the procedure. Optionally, additional labels define *secondary entry points*. The ENTRY statement also defines secondary entry points. For example:

```
B: ENTRY;
```

Any ENTRY statements encountered during sequential flow are not executed; control flows around the ENTRY statement.

The syntax for PROCEDURE and ENTRY statements is shown under "PROCEDURE and ENTRY statements."

A label need not appear after the keyword END in the END statement, but if one does appear, it must match the label of the PROCEDURE statement to which the END statement corresponds (although there are exceptions—see "END statement" on page 184).

Both internal and external procedures are normally loaded into main storage at the same time as the invoking procedure. However, an external procedure can be compiled separately from the invoking procedure and loaded when needed. By the use of FETCH and RELEASE statements, the invoking procedure loads the external procedure into main storage and deletes it when it is no longer required.

Using arguments and parameters generalizes a procedure, so that data whose names are not known within the procedure can, nevertheless, be operated upon.

A parameter has no storage associated with it. A parameter allows the invoked procedure to access storage allocated in the invoking procedure. A reference to a parameter in a procedure is a reference to the corresponding argument. Any change to the value of the parameter is made to the value of the argument.

However, in certain circumstances, a dummy argument is created and the value of the original argument is not changed. In these cases, a reference to the parameter is a reference to the dummy argument. The dummy argument initially has the same value as the original argument, but subsequent changes to the parameter do not affect the original argument's value.

PROCEDURE and ENTRY statements

A procedure (subroutine or function) can have one or more entry points. The primary entry point to a procedure is established by the leftmost label of the PROCEDURE statement. Secondary entry points to a procedure are established by additional labels of the PROCEDURE statement and by the ENTRY statement.

Each entry point has an entry name. See "INTERNAL and EXTERNAL attributes" on page 155 for a discussion of the rules for the creation of an external name.

Each PROCEDURE or ENTRY statement can specify its own parameters and, in the case of functions, returned value attributes.

Entry names are explicitly declared in the invoking block as entry constants for internal procedures by their presence as prefixes to PROCEDURE or ENTRY statements. It is an error to declare an internal entry name in a DECLARE statement.

If a PROCEDURE or ENTRY statement has more than one *entry-constant* the first can be considered as the only label of the statement. Each subsequent *entry-constant* can be considered as a separate ENTRY statement having an identical parameter list as specified in the PROCEDURE statement. For example:

```
A:  I:  PROCEDURE (X);
```

is the same as:

```
A:   PROCEDURE (X);
I:   ENTRY (X);
```

Since the attributes of the returned value are not explicitly stated, default attributes are supplied, and the attributes of the value returned depend on whether the procedure has been invoked as A or I.

When an EXTERNAL ENTRY is declared without a parameter descriptor list, matching between parameters and arguments does not occur. Therefore, no diagnostic message is issued if any arguments are specified in a CALL to such an entry point. For example:

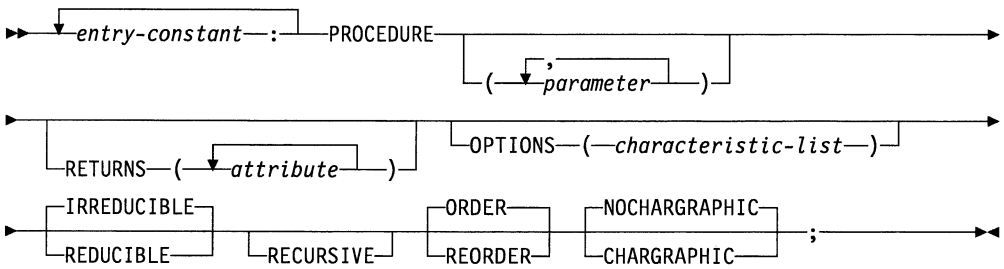
```
DECLARE X ENTRY EXTERNAL;
CALL X(parameter);
```

PROCEDURE statement

The PROCEDURE statement:

- And the corresponding END statement delimit a procedure
- Defines the primary entry point to the procedure (and, optionally, secondary entry points)
- Specifies the parameters, if any, for the primary entry point
- Can specify options that a procedure can have
- Can specify the attributes of the value returned by the procedure if it is invoked as a function at its primary entry point

The syntax for the PROCEDURE statement is:



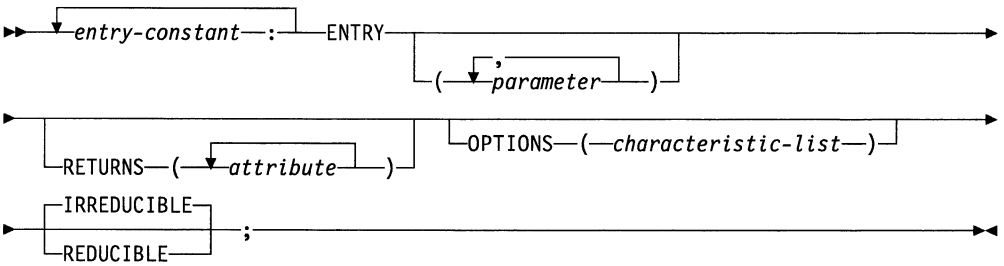
Abbreviations: CHARG for CHARGRAPHIC
IRRED for IRREDUCIBLE
NOCHARG for NOCHARGRAPHIC
PROC for PROCEDURE
RED for REDUCIBLE

The keywords can appear in any order.

ENTRY statement

The ENTRY statement specifies a secondary entry point of a procedure. The ENTRY statement must be internal to the procedure for which it defines a secondary entry point. It cannot be within a do-group that specifies repetitive execution or internal to an ON-unit.

When an ENTRY statement is encountered in sequential program flow, control passes around it. The syntax for the ENTRY statement is:



Abbreviations: IRRED for IRREDUCIBLE
RED for REDUCIBLE

The keywords can appear in any order.

parameter

Specifies that a name in an invoked procedure represents an argument passed to that procedure. A name is explicitly declared with the *parameter attribute* by its appearance in a parameter list. The name must not be subscripted or qualified. Keywords following the procedure can appear in any order.

See "Association of arguments and parameters" on page 128 and "Parameter attributes" on page 116.

RETURNS option

Specifies, for a function procedure, the attributes of the value returned by the function.

If more than one attribute is specified, they must be separated by blanks (except for attributes such as precision, that are enclosed in parentheses).

The attributes that can be specified are any of the data attributes and alignment attributes for variables (except those for ENTRY variables), as shown in Table 12 on page 23. The OFFSET attribute can include an area reference.

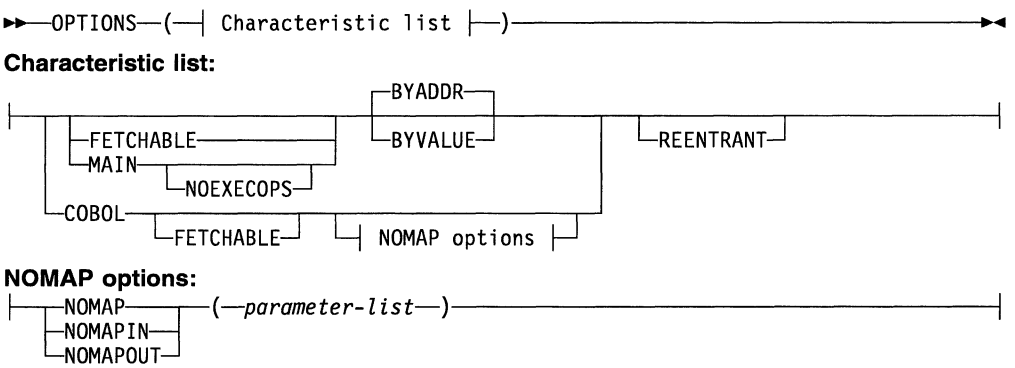
String lengths and area sizes must be specified by integers. The returned value has the specified length or size.

The RETURNS attribute must agree with the attributes specified in (or defaults for) the RETURNS option of the PROCEDURE or ENTRY statement to which the entry name is prefixed. The returned value has attributes determined from the RETURNS option. If they do not agree, there is an error, since no conversion is performed.

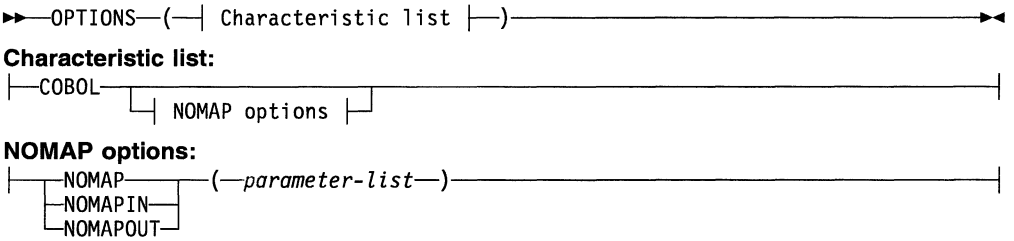
If the RETURNS option is not specified, the attributes of the returned value are determined by default (see "Defaults for data attributes" on page 159).

OPTIONS option

Specifies one or more processing characteristics. The syntax for the *characteristic-list* of a PROCEDURE statement is:



The syntax for the *characteristic-list* of an ENTRY statement is:



The options are separated by blanks or commas. They can appear in any order.

OPTIONS can be specified only for an external procedure, and only one procedure must have the OPTIONS (MAIN) designation.

Note: For information on interlanguage communication, see the *LE/VSE Programming Guide*.

The meaning of the options are:

BYADDR or BYVALUE options

specify how all parameters defined for this procedure entry are passed. If you specify BYADDR, parameters are received by address. If you specify BYVALUE, parameters are received by value. Any change to a parameter that is being passed by value is not reflected in the argument passed by the caller.

BYADDR is the default unless the procedure is MAIN and is being compiled with the SYSTEM(CICS), SYSTEM(DLI), or SYSTEM(DL1) compile-time option. In this case, BYVALUE is implied and BYADDR is invalid.

BYADDR: BYADDR can be specified for EXTERNAL PROCEDURE statements.

For example:

```
EXTR: PROCEDURE(A,B) OPTIONS(BYADDR);
      DCL (A,B) FLOAT;
```

BYVALUE: BYVALUE may be specified for EXTERNAL PROCEDURE statements. It cannot be specified for ENTRY statements and internal procedures. Specifying BYVALUE for a MAIN procedure implies the specification of NOEXECOPS.

Options BYVALUE procedures can only have scalar parameters and return values that are either POINTER or REAL FIXED BINARY(31,0). If all parameters are POINTER, the last parameter must not have the NULL pointer value. If you need to pass a "null" pointer in this situation, use the SYSNULL pointer and, if necessary, convert it to a NULL pointer.

If a procedure is compiled with SYSTEM(VSE), do not specify BYVALUE. If you specify BYVALUE, the parameter list is passed as is to the MAIN procedure.

For example:

```
EXTR: PROC(P,Q) OPTIONS(BYVALUE);
      DCL (P,Q) POINTER;
```

MAIN The PL/I procedure is the initial procedure of a PL/I program. The operating-system control program invokes it as the first step in the execution of that program.

COBOL

The PL/I procedure is invoked at this entry point by only a COBOL subprogram.

NOEXECOPS

The NOEXECOPS option is valid only with the MAIN option. It indicates that the character string is passed to the main procedure without evaluating the run-time options for the SYSTEM(VSE) compile-time option. The run-time options are not honored if they are received.

NOEXECOPS is also implied with the SYSTEM(CICS), SYSTEM(DLI), and SYSTEM(DL1) compile-time options.

For information about the interaction of NOEXECOPS and the SYSTEM compiler option, see the *LE/VSE Programming Guide*.

FETCHABLE

The FETCHABLE option indicates that an external procedure can be fetched and is entered through this entry point. If the procedure has multiple labels, the FETCHABLE option is applied to the primary entry point. When a phase containing a FETCHABLE procedure is fetched, control passes to that procedure.

If the phase has multiple FETCHABLE procedures, then the linkage editor may choose any of those procedures as the one that gets control.

The FETCHABLE option is valid only on external procedures.

Note: If a phase has both a FETCHABLE and a MAIN procedure, control passes to the FETCHABLE procedure when the phase is fetched, and to the MAIN procedure when the phase is executed by job control.

NOMAP, NOMAPIN, and NOMAPOUT

The mapping of a data aggregate passed by COBOL might not match the mapping used by PL/I. If PL/I detects a possible mapping discrepancy, PL/I creates a dummy argument for the data aggregate. The dummy argument uses the PL/I mapping algorithm. PL/I then automatically assigns the incoming data aggregate values to the dummy argument. The PL/I routine uses the dummy argument. On return to COBOL routine, the dummy argument values are remapped to the data aggregate.

You can use the NOMAP, NOMAPIN and NOMAPOUT options to prevent this automatic mapping of values:

NOMAP Specifies no mapping of values either at entry to PL/I or on return to COBOL. When you specify this option, PL/I does not create a dummy argument.

NOMAPIN Specifies no mapping of values at entry to PL/I. When you specify this option, PL/I creates a dummy argument.

NOMAPOUT Specifies no mapping of values on return to COBOL. When you specify this option, PL/I creates a dummy argument.

The NOMAP, NOMAPIN and NOMAPOUT options are effective only for structures that are passed from COBOL. The options do not apply to scalars.

Use these options when program efficiency is important. These options help you avoid unnecessary mapping code.

Note: NOMAP, NOMAPIN, and NOMAPOUT can all appear in the same OPTIONS specification. Specifying NOMAPIN and NOMAPOUT for the same parameter is the same as specifying NOMAP for that parameter.

For information about interlanguage communication, see the *LE/VSE Programming Guide*.

parameter-list

The parameter-list is the name of the parameters, already specified in the PROCEDURE or ENTRY statement, to which the option NOMAP, NOMAPIN or NOMAPOUT option is to be applied. If you do not specify a list, the option is applied to all parameters. You should not repeat the

same parameter name in a parameter-list. Parameters can appear in any order, and are separated by commas or blanks.

REENTRANT
For a program to be reentrant, you must specify REENTRANT and you must not do anything that alters static storage during execution. (In the latter case, no compiler error message appears).

Additional PROCEDURE and ENTRY statement options are:

IRREDUCIBLE and REDUCIBLE options
If REDUCIBLE or IRREDUCIBLE is specified, it is checked for syntax errors and ignored.

RECURSIVE option
RECURSIVE must be specified if the procedure might be invoked recursively. It applies to all of the entry points (primary and secondary) that the procedure has.

ORDER and REORDER options
ORDER and REORDER are optimization options that are specified for a procedure or begin block. If neither option is specified for the external procedure, ORDER is the default. The default for internal blocks is to inherit ORDER or REORDER from the containing block.

The ORDER option 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.

The ORDER and REORDER options are discussed in more detail in the *PL/I VSE Programming Guide.*

CHARGRAPHIC and NOCHARGRAPHIC options
This option does not require, or preclude, the use of the GRAPHIC compiler option.

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 mixed character assignments.
- STRINGSIZE condition causes MPSTR to be called. STRINGSIZE must be enabled for character assignments that can cause truncation. (For information on the MPSTR built-in function see "MPSTR — String-handling" on page 385.) For example:

```
NAME: PROCEDURE 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.

Parameter attributes

A name is explicitly declared with the parameter attribute by its appearance in the parameter list of a PROCEDURE or ENTRY statement. Attributes other than parameter can be supplied by a DECLARE statement internal to the procedure. If attributes are not supplied in a DECLARE statement, default attributes are applied. Table 12 on page 23, and the following discussion, describe the attributes that can be declared for a parameter.

A parameter always has the attribute INTERNAL. It must be a level-one name.

Because a parameter has no associated storage within the invoked procedure, it cannot be declared to have any of the storage attributes STATIC, AUTOMATIC, BASED, or DEFINED. However, it can be declared to have the CONTROLLED attribute. Thus, there are two classes of parameters, as far as storage allocation is concerned: those that have no storage class (*simple parameters*) and those that have the CONTROLLED attribute (*controlled parameters*). Only controlled parameters can have the INITIAL attribute.

Parameters used in record-oriented input/output, or as the base variable for DEFINED items, must be in connected storage. If such a parameter is an aggregate, it must have the CONNECTED attribute, both in its declaration in the procedure, and, where applicable, in the descriptor list of the procedure entry declaration.

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 length and size of an area or string parameter, must be the same as the current generation of the corresponding argument.

Simple parameter bounds, lengths, and sizes

Bounds, lengths, and sizes of simple parameters must be specified either by asterisks or by constants. When the actual length, bounds, or size may be different for different invocations, each can be specified in a DECLARE statement by an asterisk. When an asterisk is used, the length, bounds, or size are taken from the current generation of the associated argument.

An asterisk is not allowed as the length specification of a string that is an element of an aggregate, if the associated argument creates a dummy. The string length must be specified as an integer.

Controlled parameter bounds, lengths, and sizes

The bounds, length, or size of a controlled parameter can be specified in a DECLARE statement either by asterisks or by element expressions.

Asterisk notation: When asterisks are used, length, bounds, or size of the controlled parameter are taken from the current generation of the associated argument. Any subsequent allocation of the controlled parameter uses these same bounds, length, or size, unless they are overridden by a different length, bounds, or size specification in the ALLOCATE statement. If no current generation of the argument exists, the asterisks determine only the dimensionality of the parameter, and an ALLOCATE statement in the invoked procedure must specify bounds, length, or size for the controlled parameter before other references to the parameter can be made.

Expression notation: Each time the parameter is allocated, the expressions are evaluated to give current bounds, lengths, or sizes for the new allocation. However, such expressions in a DECLARE statement can be overridden by a bounds, length, or size specification in the ALLOCATE statement itself.
For example:

```
MAIN: PROCEDURE OPTIONS(MAIN);
      DECLARE (A(20), B(30), C(100),
               D(100))CONTROLLED,
               NAME CHARACTER (20),
               I FIXED(3,0);
      DECLARE (SUB1,SUB2) ENTRY;

      ALLOCATE A,B;
      CALL SUB1(A,B);
      FREE A,B;
      FREE A,B;

      GET LIST (NAME,I);
      CALL SUB2 (C,D,NAME,I);
      FREE C,D;
      END MAIN;

*PROCESS;
    SUB1: PROCEDURE (U,V);
          DCL (U(*), V(*)) CONTROLLED;

          ALLOCATE U(30), V(40);
          RETURN;
          END SUB1;

*PROCESS;
    SUB2: PROCEDURE (X,Y,NAMEA,N);
          DECLARE (X(N),Y(N))CONTROLLED,
                  NAMEA CHARACTER (*),
                  N FIXED(3,0);

          ALLOCATE X,Y;
          RETURN;
          END SUB2;
```

When SUB1 is invoked, A and B, which have been allocated as declared, are passed. The ALLOCATE statement in SUB1 specifies bounds for the arrays; consequently, the allocated arrays, which are actually a second generation of A and B, have bounds different from the first generation. If no bounds were specified in the ALLOCATE statement, the bounds of the first and the new generation are identical.

On return to MAIN, the first FREE statement frees the second generation of A and B (allocated in SUB1), and the second FREE statement frees the first generation (allocated in MAIN).

In SUB2, X and Y are declared with bounds that depend upon the value of N. When X and Y are allocated, this value determines the bounds of the allocated array.

The asterisk notation for the length of NAMEA indicates that the length is to be picked up from the argument (NAME).

Procedure activation

A procedure can be invoked at any point at which an entry name of the procedure is known. Execution of the invoked procedure can be only synchronous; that is, the execution of the invoking procedure is suspended until control is returned to it.

Sequential program flow passes around the 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 106 tells how to activate the main procedure.

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

- After the keyword CALL in a CALL statement (as described in "CALL statement")
- After the keyword CALL in the CALL option of the INITIAL attribute (as described under "INITIAL attribute" on page 230)
- As a function reference (as described in "Functions")

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

When a procedure reference is executed, the procedure containing the specified entry point is activated and is said to be *invoked*. Control is transferred to the specified entry point. 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 block it invokes.

Whenever a procedure is invoked at its primary entry point, execution begins with the first statement in the invoked procedure. When a procedure is invoked at a secondary entry point, execution begins with the first statement following the statement that defines that secondary entry point. The environment established on entry to a block at the primary entry point is identical to the environment established when the same block is invoked at a secondary entry point.

Communication between two procedures is by means of arguments passed from an invoking procedure to the invoked procedure, by a value returned from an invoked procedure, and by names known within both procedures. Therefore, a procedure can operate upon different data when it is invoked from different points. For example:

```
A:  READIN:  PROCEDURE;
             statement-1
             statement-2
       ERRT: ENTRY;
             statement-3
             statement-4
             statement-5
NEXT: RETR:  ENTRY;
             statement-6
             ...
             END READIN;
```

In the example, *A* is the primary entry point. A and READIN specify the same entry point, as do NEXT and RETR. The procedure can be activated by any of the following statements:

```
CALL A;
CALL ERRT;
CALL NEXT;
CALL RETR;
CALL READIN;
```

The statement CALL A invokes procedure A at its primary entry point, and execution begins with statement-1; the statement CALL ERRT invokes procedure A at the secondary entry point ERRT, and execution begins with statement-3. Either of the statements, CALL NEXT or CALL RETR, invokes procedure A at its other secondary entry point, and execution begins with statement-6.

Alternatively, the appropriate entry value can be assigned to an entry variable that is used in a procedure reference. In the following example, the two CALL statements have the same effect:

```
DECLARE ENT1 ENTRY VARIABLE;
ENT1 = ERRT;
CALL ENT1;
CALL ERRT;
```

Procedure termination

A procedure is terminated when, by some means other than a procedure reference, control passes back to the invoking block or to some other active block.

Normal procedure terminations are:

- 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 one of the other forms of procedure references (that is, a CALL option or 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.

Abnormal procedure terminations are:

- Control reaches a GO TO statement that transfers control out of the procedure. This is not allowed if the procedure is invoked by the CALL option of the INITIAL attribute. The GO TO statement can specify a label in a containing block (the label must be known within the procedure), or it can specify a parameter that has been associated with a label argument passed to the procedure.
- A STOP or EXIT statement is executed terminating execution of the program.
- 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.

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. For 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 option specified in its PROCEDURE statement. This option also applies to any secondary entry points that the procedure might have.

The environment (values of automatic variables, etc.) 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 203). 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, a GO TO statement naming that variable in another invocation restores the environment that existed when the assignment was performed.

The environment of a procedure invoked using an entry variable from within a recursive procedure is the environment 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. For 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 of variables is also of importance in the discussion of controlled variables.)

Dynamic loading of an external procedure

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 is 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.

The appearance of an entry constant in a FETCH or RELEASE statement indicates that the procedure containing that entry constant will need to be loaded into main storage before it can be executed, unless a copy already exists in main storage. When a FETCH statement is executed, the procedure is loaded from auxiliary storage into main storage. In addition, when a CALL statement or option or a function reference is executed, the procedure is loaded into main storage. Thus, a procedure may be loaded from auxiliary storage by either:

- Execution of a FETCH statement.
- Execution of a CALL statement, or CALL option of an INITIAL attribute, or a function reference, provided that the name of the entry point of the procedure appears, somewhere in the calling procedure, in a FETCH or RELEASE statement. It is not necessary that control passes through a FETCH or RELEASE statement, either before or after execution of the CALL or function reference.

In neither case is it an error if the procedure has already been loaded into main storage.

Whichever statement loaded the procedure, execution of the CALL statement or option or the function reference invokes the procedure in the normal way.

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 the RELEASE statement.

FETCH and RELEASE restrictions

When using dynamically-loaded procedures:

1. Only external procedures can be fetched.
2. PL/I OPTIONS(MAIN) procedures cannot be fetched.
3. Fetched PL/I procedures must have been compiled by PL/I VSE.
4. Variables with the EXTERNAL attribute are not allowed in a fetched procedure.
5. Variables with the CONTROLLED attribute are not allowed in a fetched procedure unless they are parameters.
6. With the exception of the file SYSPRINT, variables with the FILE attribute are not allowed in a fetched procedure unless they are parameters. This means any other file used in the fetched procedure, including the file SYSIN, must be passed from the calling procedure.

   A file that is explicitly opened in a fetched procedure must be explicitly closed in that procedure before the procedure ends.

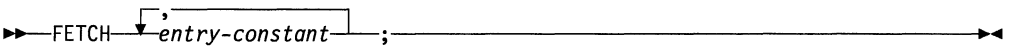
   A file that is implicitly opened in a fetched procedure must be closed only in the fetching procedure. The close must be prior to releasing the fetched procedure.

   A file that is open when it is passed to a fetched procedure must not be closed in the fetched procedure.
7. Storage for STATIC variables in the fetched procedure is allocated when the FETCH statement is executed, and is freed when a corresponding RELEASE statement is executed. Each time a procedure is fetched into main storage, a STATIC variable either is given the value specified in an INITIAL attribute, or, if there is no INITIAL attribute, is not initialized.
8. The FETCH, RELEASE, and CALL statements must specify entry constants. An entry constant for a fetched procedure cannot be assigned to an entry variable.
9. Fetched procedures cannot fetch further procedures.

Violation of the above restrictions may cause random errors. For information about other restrictions related to interlanguage communication, see the *LE/VSE Programming Guide.*

FETCH statement

The FETCH statement checks main storage for the named procedures. The named procedures must not be internal procedures. Procedures not already in main storage are loaded from auxiliary storage. Refer to the *LE/VSE Programming Guide* for more information about FETCH. The syntax for the FETCH statement is:

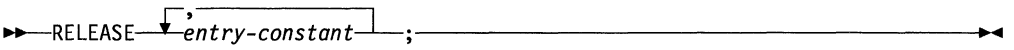


entry-constant
specifies the name by which the procedure to be fetched is known to the operating system. Details of the linkage-editing required for fetchable procedures are given in the *LE/VSE Programming Guide*.

The entry-constant must be the same as the one used in the corresponding CALL statement, CALL option, or function reference.

RELEASE statement

The RELEASE statement frees for other purposes main storage occupied by procedures identified by the 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 auxiliary storage:

```
PROG:  PROCEDURE;

       FETCH PROGA;
       CALL PROGA;
       RELEASE PROGA;

       CALL PROGB;
       GO TO FIN;

       FETCH PROGB;
FIN:  END PROG;
```

PROGA is loaded into main storage by the first FETCH statement, and executes when the first CALL statement is reached. Its storage is released when the RELEASE statement is executed. PROGB is loaded when the second CALL statement is reached, even though the FETCH statement referring to this procedure is never executed, and the same CALL statement initiates execution of PROGB. 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 link-edited 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 in auxiliary storage.

Subroutines

A *subroutine* is a procedure that is invoked by a CALL statement or CALL option of an INITIAL attribute. It can be either an external or an internal procedure.

Whenever a subroutine is invoked, the arguments of the invoking statement are associated with the parameters of the entry point, then control is passed to that entry point. The subroutine is activated, and execution of the procedure can begin.

Upon normal termination of a subroutine, by a RETURN statement or by control reaching the procedure's END statement, control is returned to the invoking block. A subroutine can be abnormally terminated as described in "Procedure termination" on page 119.

Examples

The following examples show how to invoke subroutines that are internal to and external to the invoking block.

```
PRMAIN:  PROCEDURE;
         DECLARE NAME CHARACTER (20),
         ITEM BIT(5),
         OUTSUB ENTRY;
         CALL OUTSUB (NAME, ITEM);
END PRMAIN;

*PROCESS;
OUTSUB:  PROCEDURE (A,B);
         DECLARE A CHARACTER (20),
         B BIT(5);
         PUT LIST (A,B);
END OUTSUB;
```

The CALL statement in PRMAIN invokes the procedure OUTSUB, and the argument list in this procedure reference contains the two arguments being passed to OUTSUB. The PROCEDURE statement defining OUTSUB associates two parameters, A and B, with the passed arguments. When OUTSUB is executed, each reference to A in OUTSUB is treated as a reference to NAME and each reference to B is treated as a reference to ITEM. Therefore, the PUT LIST (A,B) statement transmits the values of NAME and ITEM to the output file, SYSPRINT. In the declaration of OUTSUB within PRMAIN, no parameter descriptor has to be associated with the ENTRY attribute, since the attributes of NAME and ITEM match those of, respectively, A and B.

```
A:  PROCEDURE;
    DECLARE RATE FLOAT (10),
            TIME FLOAT(5),
            DISTANCE FLOAT(15),
            MASTER FILE;
    CALL READCM (RATE, TIME,
         DISTANCE, MASTER);

READCM:
    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;
```

The arguments RATE, TIME, DISTANCE, and MASTER are passed to the procedure READCM and associated with the parameters W, X, Y, and Z. Consequently, in the subroutine, 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.

Built-in subroutines

You can make use of *built-in subroutines*. These have entry names that are defined at compile-time and are invoked by a CALL statement. The entry names are known as *built-in names*, and can be explicitly or contextually declared to have the BUILTIN attribute.

The use of these subroutines is described in the *PL/I VSE Programming Guide* unless otherwise noted. The facilities and the built-in names are as follows:

- The subroutine PLICKPT writes a checkpoint record.
- The subroutine PLIREST requests a restart.
- The subroutine PLICANC causes the system to issue a message; the program continues normal operation. (In some implementations, PLICANC cancels restart activity from any checkpoints taken. However, VSE does not support this function.)
- The sort/merge subroutines are PLISRTA, PLISRTB, PLISRTC, and PLISRTD.
- The subroutine PLIDUMP provides a formatted dump of main storage. PLIDUMP is discussed in the *LE/VSE Debugging Guide and Run-Time Messages.*
- The subroutine PLIRETC allows you to set the return code of your program.
- The subroutine PLITDLI is used for DL/I applications. This is described in *DL/I DOS/VS Application Programming: CALL and RQDLI Interfaces.*

For additional DL/I considerations, see the *LE/VSE Programming Guide.*

Note: PLITDLI cannot be declared with the BUILTIN attribute but is treated as a special subroutine.

Functions

A *function* is a procedure that is invoked by a function reference in an expression. A *function reference* is an entry reference that represents an entry name (a particular entry point of a procedure) invoked as a function. A function returns a value, and control, to replace the function reference in the evaluation of the expression in which the function reference appears. This single value can be of any data type except entry.

Whenever a function is invoked, the arguments of the invoking statement are associated with the parameters of the entry point, and control is then passed to that entry point. The function is activated, and execution of the procedure can begin.

The RETURN statement terminates a function and returns control to the invoking procedure. Its use in a function differs somewhat from its use in a subroutine; in a function, not only does it return control but it also returns a value to the point of invocation.

A function can be abnormally terminated as described in "Procedure termination" on page 119. 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 147 for additional information.

Examples

The following examples illustrate the invocation of functions that are internal to and external to the invoking block.

In the following example, the assignment statement contains a reference to a function SPROD:

```
MAINP: PROCEDURE;
       GET LIST (A, B, C, Y);
       X = Y**3+SPROD(A,B,C);

SPROD: PROCEDURE (U,V,W)
         RETURNS (BIN FLOAT(21));
       DCL (U,V,W) BIN FLOAT(53);
       IF U > V + W
               THEN RETURN (0);
               ELSE RETURN (U*V*W);
       END SPROD;
```

When SPROD is invoked, the arguments A, B, and C are associated with the parameters U, V, and W, respectively. 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.

In the following example, when TPROD is invoked, LAB1 is associated with parameter Z. If U is greater than V + W, control returns to MAINP at the statement labeled LAB1, and evaluation of the expression that invoked TPROD is discontinued. If U is not greater than V + W, a return to MAINP is made in the normal fashion.

```
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);
        X = Y**3+TPROD(A,B,C,LAB1);
LAB1:   CALL ERRT;
        END MAINP;

*PROCESS;
TPROD:  PROCEDURE (U,V,W,Z)
          RETURNS (BIN FLOAT(21));
         DCL (U,V,W) BIN FLOAT(53);
         DECLARE Z LABEL;

         IF U > V + W
               THEN GO TO Z;
               ELSE RETURN (U*V*W);
         END TPROD;
```

Built-in functions

Besides function references to procedures written by the programmer, a function reference can invoke one of a set of *built-in functions*. Each built-in function is described in Chapter 16, "Built-in functions, subroutines, and pseudovariables" on page 360.

Built-in functions include the commonly used arithmetic functions and others, such as functions for manipulating strings and arrays.

You invoke built-in functions 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.

Some built-in functions are compiled as in-line code rather than as procedure invocations. Functions that are compiled as in-line code do not have entry values. In fact, only the mathematical built-in functions are defined with entry values. Thus, reference to the nonmathematical built-in functions cannot be used in contexts requiring entry values.

Association of arguments and parameters

When a function or subroutine is invoked, parameters in the parameter list are associated, from left to right, with the arguments in the argument list. The number of arguments and parameters must be the same.

In general:

- Problem data arguments can be passed to parameters of any problem data type, except that graphic values can only be passed to graphic parameters, and graphic parameters must have graphic arguments.
- Program control data arguments must be passed to parameters of the same type.
- Aggregate parameters can have aggregate or element arguments.

Expressions in the argument list are evaluated in the invoking block, before the subroutine or function is invoked.

Dummy arguments

A reference to an argument, not its value, is generally passed to a subroutine or function. This is known as passing arguments by reference. However, this is not always possible or desirable. Constants, for example, should not be altered by an invoked procedure. Therefore, the compiler allocates storage (in storage belonging to the invoking procedure) for some arguments using attributes that agree with the parameter, converts, and assigns to the allocated storage, and then passes a reference to the allocated storage. These storage locations are called *dummy arguments.* Any change to a parameter for which a dummy argument has been created is reflected only in the value of the dummy argument and not in the value of the original argument from which it was constructed.

Deriving dummy argument attributes

This is performed for internal entry constants using the declarations of the parameters; but for entry variables and external entry constants, a parameter descriptor list must be given in an appropriate entry declaration if conversion is required. The attributes of a dummy argument are derived 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.
- For the bounds of an array, the length of a string or the size of an area, if specified by asterisk notation in the parameter declaration, from the bound, length or size of the argument itself.

In all other cases, a reference to the argument is passed (in effect, the storage address of the argument is passed).

The parameter becomes identical with the passed argument; thus, changes to the value of a parameter will be reflected in the value of the original argument only if a dummy argument is not passed.

Dummy argument restrictions

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.

If a parameter is an *array*, the argument can be an array expression or an element expression. If the argument is an element expression, the corresponding parameter descriptor or declaration must specify the bounds of the array parameter,

as integers. This causes the construction of a dummy array argument, whose bounds are those of the array parameter. The value of the element expression is then assigned to the value of each element of the dummy array argument.

If the argument is an array expression, the number of dimensions must be the same, and the bounds must either be the same or must, for the parameter, be declared with asterisks.

If a parameter is a *structure*, the argument must be a structure expression or an element expression. If the argument is an element expression, the corresponding parameter descriptor for an external entry point must specify the structure description of the structure parameter (only level numbers need be used—see the discussion of the "ENTRY attribute" on page 136, for details). This causes the construction of a dummy structure argument, whose description matches that of the structure parameter. The value of the element expression then becomes the value of each element of the dummy structure argument. The relative structuring of the argument and the parameter must be the same; the level numbers need not be identical. The element value must be one that can be converted to conform with the attributes of all the elementary names of the structure.

If the parameter is an *array of structures*, the argument can be an element expression, an array expression, a structure expression, or an array of structures expression.

Whenever a varying-length element string argument is passed to a nonvarying element string parameter whose length is undefined (that is, specified by an asterisk), a dummy argument whose length is the current length of the original argument is passed to the invoked procedure.

When the argument is a varying-length string array passed to a nonvarying undefined-length array parameter, a dummy argument whose element length is the maximum length is passed.

If the parameter has one of the *program control data* (except locator) attributes, the argument must be a reference of the same data type.

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 attribute and references" on page 144, for a description of generic name arguments for entry parameters.

If a parameter is a *locator* of either pointer or offset type, the argument must be a locator reference of either type. If the types differ, a dummy argument is created. The parameter descriptor of an offset parameter must not specify an associated area.

A *simple 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.

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 these generations. Thus, 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, thus allowing the manipulation of the allocation stack of the associated argument.

When no parameter descriptor is given, the entire stack is passed. In this case, the parameter can be simple or controlled and be correspondingly associated with either the latest generation or the entire stack.

In addition, a dummy argument is created when the original 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 simple parameters when only bounds, lengths, or size differ and, for the parameter, these are declared with asterisks.

  This does not apply when an expression other than an integer is used to define the bounds, length or size of a controlled parameter. The compiler assumes that the argument and parameter bounds, length or size 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
- An iSUB-defined array

Passing an argument to the MAIN procedure

The PROCEDURE statement and ENTRY statements for the main procedure can have parameter lists. 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 or subsystems supported by PL/I, a single argument is passed to the program. See the *PL/I VSE Programming Guide* for a description of the SYSTEM compile-time option. If this facility is used, the parameter must be declared as a VARYING character string. The current length is set equal to the argument length at run-time. For example:

```
TOM:  PROC (PARAM) OPTIONS (MAIN);
      DCL PARAM CHAR(100) VARYING;
```

Storage is allocated only for the current length of the argument. The source program will overwrite adjacent information if a value greater than the current length is assigned to the parameter.

When NOEXECOPS is specified, the MAIN procedure can have one of the following as parameters:

- A single parameter that is a VARYING CHARACTER string. The parameter is passed as is, and a descriptor is set up. ("/", if contained in the string, is treated as part of the string.) The following example:

  ```
  MAIN:PROC(PARM) OPTIONS(MAIN NOEXECOPS);
       DCL PARM CHAR(n) VARYING;
  ```

  Shows a MAIN procedure that can be invoked as follows:

  ```
  // EXEC MAIN,PARM='REPORT,LIST'
  ```

  The PARM contains REPORT,LIST and has a length of 11.

- Other parameters (such as, more than one parameter or a single parameter that is not a VARYING CHARACTER string). The parameter list is passed as is, and no descriptors are set up. The caller of the PL/I MAIN procedure must know what is expected by the procedure, including any required descriptors. The following example:

  ```
  MAIN:PROC(FUNC, P) OPTIONS(MAIN NOEXECOPS);
       DCL FUNC FIXED BIN(31);
       DCL P PTR;
  ```

  Shows a MAIN procedure that can be invoked from an assembler program. The assembler program should set register 1 to point to the parameter list prior to linking to the PL/I procedure.

Begin blocks

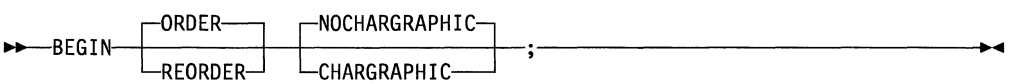
A begin block is a sequence of statements delimited by a BEGIN statement and a corresponding END statement. For example:

```
B:  BEGIN;
    statement-1
    statement-2
       .
       .
       .
    statement-n
    END B;
```

Unlike a procedure, a label is optional for a begin block. If one or more labels are prefixed to a BEGIN statement, they serve only to identify the starting point of the block. The label following END is optional. (There are exceptions; see "END statement" on page 184).

BEGIN statement

The BEGIN statement and a corresponding END statement delimit a begin-block. The syntax for the BEGIN statement is:



The options of the BEGIN statement can appear in any order.

ORDER and REORDER options
The ORDER and REORDER options are the same as those that you can specify with the PROCEDURE statement, and are described under "PROCEDURE and ENTRY statements" on page 109.

CHARGRAPHIC and NOCHARGRAPHIC options
The CHARGRAPHIC and NOCHARGRAPHIC options are also the same as those used with PROCEDURE statement, and are described under "PROCEDURE and ENTRY statements" on page 109.

Begin-block activation

Begin blocks are activated through sequential flow or as a *unit* in an IF, ON, WHEN, or OTHERWISE statement.

Control can be transferred to a labeled BEGIN statement by execution of a 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; that is, when:

- Control reaches the END statement for the block. When this occurs, control moves to the statement physically following the END, except when the block is an ON-unit.
- The execution of a GO TO statement within the begin block (or any block activated from within that begin block) transfers control to a point not contained within the block.
- A STOP or EXIT statement is executed.
- Control reaches a RETURN statement that transfers control out of the begin block (and out of its containing procedure as well).

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 119.

Entry data

Entry data can be an entry constant or the value of an entry variable.

An entry constant is a name written as a label prefix to a PROCEDURE or ENTRY statement, or a name declared with the ENTRY attribute and not the VARIABLE attribute, or the name of a mathematical built-in function.

An entry constant can be assigned to an entry variable. For example:

```
P:  PROCEDURE;
DECLARE EV ENTRY VARIABLE,
   (E1,E2) ENTRY;

EV = E1;
CALL EV;
EV = E2;
CALL EV;
```

P, E1, and E2 are entry constants. EV is an entry variable. The first CALL statement invokes the entry point E1. The second CALL invokes the entry point E2.

The following example contains a subscripted entry reference:

```
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;
```

The five entries A, B, C, D, and E are each invoked with the parameters X, Y, and Z.

When an entry constant which is an entry point of an internal procedure is assigned to an entry variable, the assigned value remains valid only for as long as the block that the entry constant was internal to remains active (and, for recursive procedures, current).

ENTRYADDR built-in function and pseudovariable allows you to get or set the address of the entry point of a PROCEDURE or an ENTRY in an entry variable. The address of the entry point is the address of the first instruction that would be executed if the entry were invoked. For example:

```
PTR1 = ENTRYADDR(ENTRY_VBL);
```

obtains the address of the entry point, and

```
ENTRYADDR(ENTRY_VBL) = PTR2;
```

sets the address of the entry point.

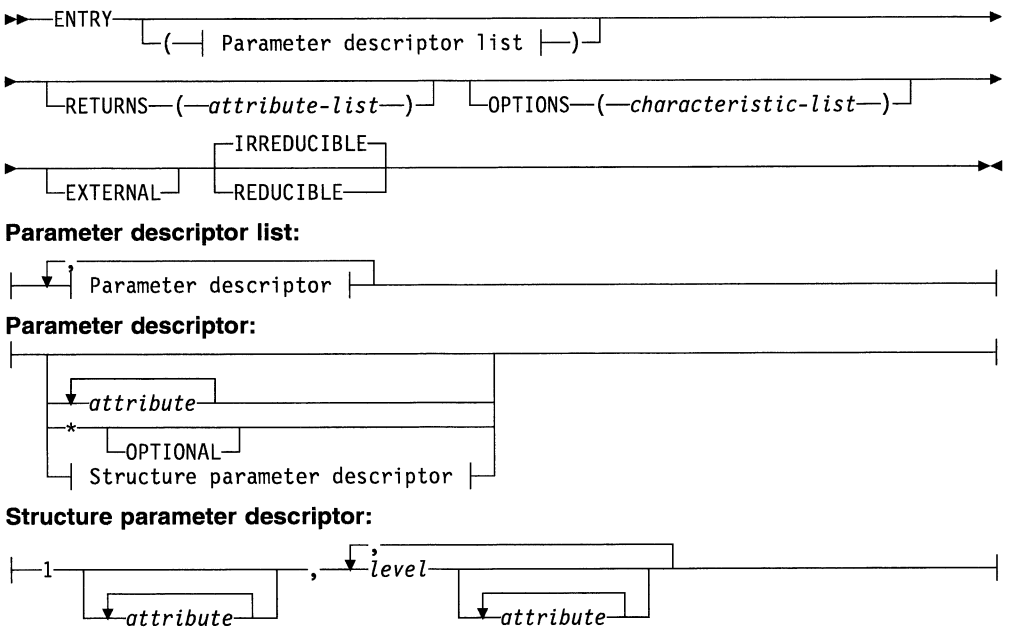
Declaring entry data

Internal entry constants are explicitly declared by the appearance of a label prefix to a PROCEDURE or ENTRY statement. 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
- Optionally specifies the attributes of the value that is returned by the procedure if the entry is invoked as a function

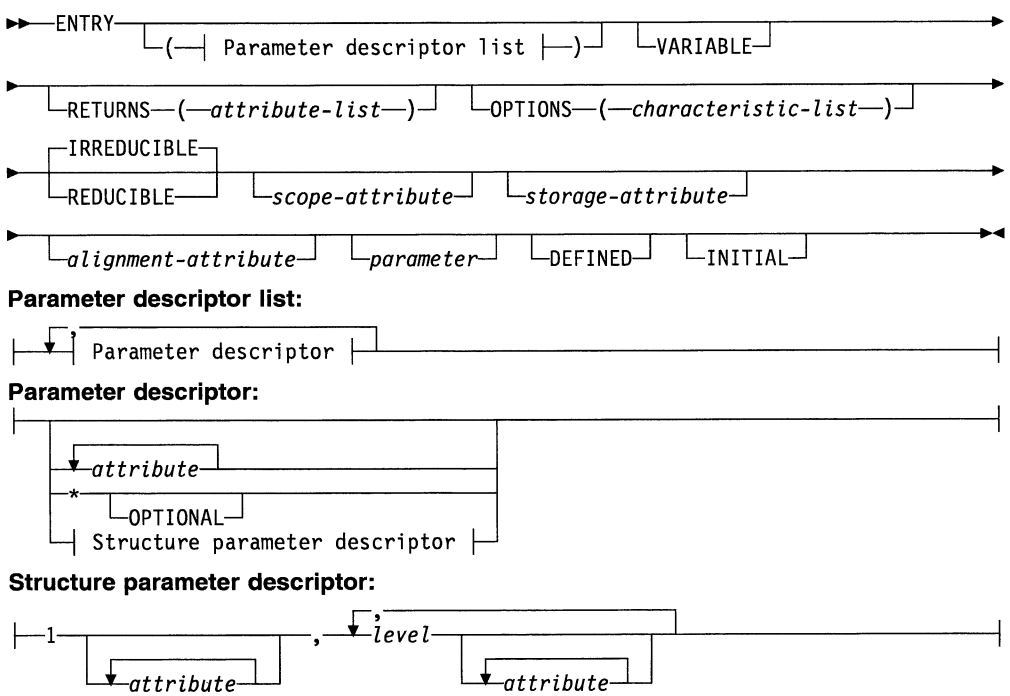
The possible attributes of the declaration of an external entry constant are shown in the following syntax diagram:



The attributes can appear in any order.

Entry variable

The possible attributes of the declaration of an entry variable (which can contain both internal and external entry values) are listed below. The variable can be part of an aggregate; structuring and dimension attributes are not shown.



The attributes can appear in any order.

VARIABLE must be specified or implied.

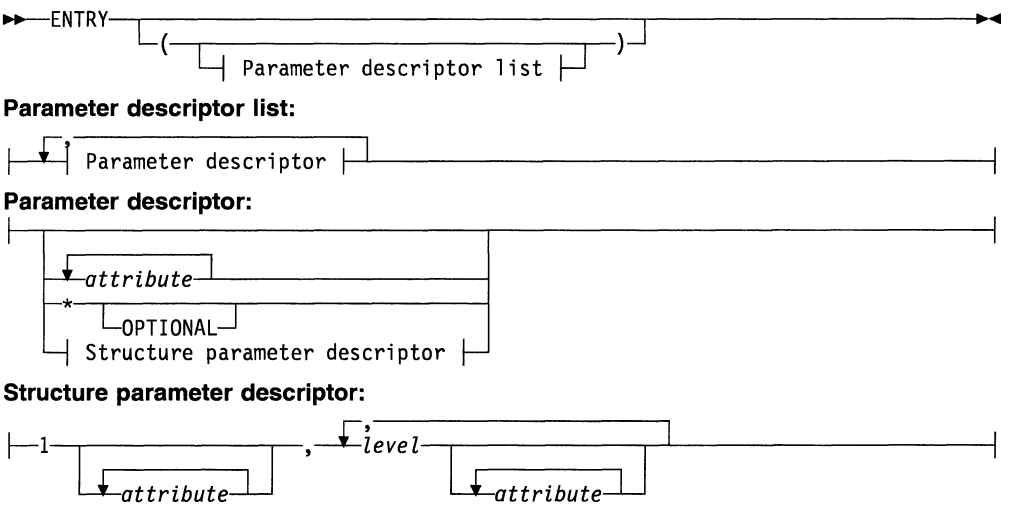
An ENTRY can be BASED, but a based ENTRY cannot be used as (or as part of) an argument of a CALL statement.

The ENTRYADDR built-in function and pseudovariable can be used to manipulate entry point addresses of procedures.

Scope, storage, and alignment attributes are described in other chapters.

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:



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 the argument attributes 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 the argument matches the parameter.

If a descriptor is not required for a parameter, the absence of the descriptor must be indicated by a comma or an asterisk. For example:

```
ENTRY(CHARACTER(10),,,FIXED DEC)       indicates four parameters

ENTRY(*)                               indicates one parameter

ENTRY(FLOAT BINARY,)                   indicates two parameters

ENTRY( )      specifies that the entry name must never have any arguments
```

The attributes can appear in any order in a *parameter-descriptor,* but for an array parameter-descriptor, the dimension attribute must be the first specified.

For a *structure-parameter-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 to a *parameter-descriptor* unless attributes or level numbers are specified in the descriptor. If a level number and/or the dimension attribute only is specified in a descriptor, FLOAT DECIMAL(6) REAL are the defaults. Defaults can be specified by use of the DESCRIPTORS option of the DEFAULT statement.

Defaults are not applied if an asterisk is specified. For example, in the following declaration, defaults are applied only to the third parameter.

```
DCL X ENTRY(*, * OPTIONAL, ALIGNED); /* DEFAULTS APPLIED TO 3RD PARM */
```

Extents (lengths, sizes, and bounds) in *parameter-descriptors* must be specified by integers or by asterisks. Extents in descriptors for controlled parameters must be specified by asterisks.

The ENTRY attribute, without a *parameter-descriptor-list,* is implied by the the attributes OPTIONS and RETURNS.

The maximum allowable depth of nesting of the ENTRY attribute is two. For example:

```
DCL E ENTRY(ENTRY(FIXED));
```

is allowed, but:

```
DCL E ENTRY(ENTRY(ENTRY(FIXED)));
```

is not allowed.

The descriptors for the parameters in the following example:

```
TEST:PROCEDURE (A,B,C,D,E,F);

   DECLARE A FIXED DECIMAL (5),
           B FLOAT BINARY (15),
           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;
```

could be declared as follows:

```
DECLARE TEST ENTRY
            (DECIMAL FIXED (5),
            BINARY FLOAT (15),
           ,
           1,
            2,
            2,
             3 DECIMAL FIXED,
             ,
           (4) CHARACTER (10));
```

OPTIONAL attribute

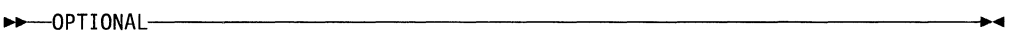
The OPTIONAL attribute can be specified in the parameter-descriptor-list of the ENTRY attribute. The ENTRY must have the OPTIONS(ASSEMBLER) attribute.

OPTIONAL arguments can be omitted in calls by specifying an asterisk for the argument. The parameter descriptor corresponding to the omitted argument must have the OPTIONAL attribute. An omitted item can be anywhere in the argument list, including at the end.

You cannot specify OPTIONAL in the declaration of a parameter, in the DEFAULT statement, or as a generic-descriptor attribute. You also cannot:

- Apply OPTIONAL and BYVALUE to the same parameter
- Omit arguments of generic names or built-in names

The syntax for the OPTIONAL attribute is:



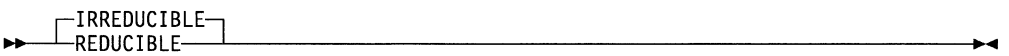
For example:

```
DCL X ENTRY (FLOAT OPTIONAL,
             FLOAT OPTIONAL)
             EXTERNAL OPTIONS (ASM);
DCL F FLOAT;

CALL X (*, *);        /*  BOTH ARGUMENTS ARE OMITTED */
CALL X (*, F);        /*  FIRST ARGUMENT IS OMITTED  */
CALL X (F, *);        /*  LAST ARGUMENT IS OMITTED   */
```

Note: An omitted argument is indicated by a word of zeros in the argument list. If the omitted argument is also the last argument, the high order bit of the zero word is set to '1'B.

IRREDUCIBLE and REDUCIBLE attributes

If the REDUCIBLE or IRREDUCIBLE attributes are specified in your program, they are checked for syntax errors, and the implied attribute ENTRY is applied; they are otherwise ignored. The syntax for the IRREDUCIBLE and REDUCIBLE attributes is:



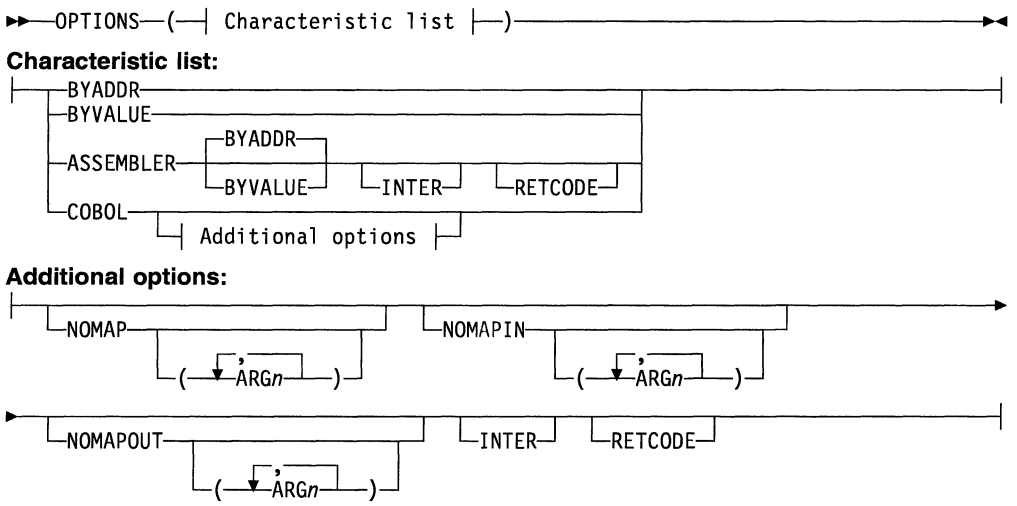
Abbreviations: IRRED for IRREDUCIBLE
RED for REDUCIBLE

OPTIONS attribute

The OPTIONS attribute specifies options that an entry point can have, similar to the OPTIONS option of PROCEDURE and ENTRY statements.

The OPTIONS attribute is required if the entry point is for a routine that is written in COBOL or assembler language. The COBOL, ASSEMBLER, and *additional-options* are described only briefly below. For information on interlanguage communication, see the *LE/VSE Programming Guide*.

The syntax for the OPTIONS attribute is:



where *n* is an integer.

The keywords can appear in any order.

BYADDR or BYVALUE options

specify how all arguments defined for this entry are passed. If you specify BYADDR, parameters are received by address. If you specify BYVALUE, parameters are received by value. Any change to a parameter that is being passed by value is not reflected in the argument passed by the caller.

BYADDR is the default.

For example:

```
DECLARE F ENTRY(FIXED BIN, PTR, CHAR(4))
   OPTIONS(BYADDR);
```

BYVALUE: BYVALUE entry points can only have scalar arguments and return values that are either POINTER or REAL FIXED BINARY(31,0).

For example:

```
DECLARE EXTR ENTRY (FIXED BIN(31), PTR) OPTIONS(BYVALUE);
```

COBOL

specifies that the designated entry point is in a COBOL subprogram.

ASSEMBLER

Abbreviation: ASM

specifies that the designated entry point is in an assembler subroutine. PL/I will pass arguments directly to the subroutine, rather than via PL/I control blocks. Entries with the ASSEMBLER option are subject to the following rules:

- They cannot be used as a function reference.
- Any number of arguments can be passed in the CALL statement invoking the entry, from zero up to the number specified by the entry declaration, but intervening arguments cannot be omitted.

NOMAP, NOMAPIN, and NOMAPOUT

The mapping of a data aggregate passed by PL/I to COBOL might not match the mapping used by COBOL. If PL/I detects a possible mapping discrepancy, PL/I creates a dummy argument for the data aggregate, assigns data aggregate values to the dummy argument, and passes the dummy argument to the COBOL routine. The dummy argument uses the COBOL mapping algorithm. On return to the PL/I routine, the values in the dummy argument are remapped to the original data argument.

You can use the NOMAP, NOMAPIN and NOMAPOUT options to prevent this automatic mapping of values:

NOMAP Specifies no mapping of values either at invocation of the COBOL routine or on return to PL/I. When you specify this option, PL/I does not create a dummy argument.

NOMAPIN Specifies no mapping of values at invocation of the COBOL routine. When you specify this option, PL/I creates a dummy argument.

NOMAPOUT Specifies no mapping of values on return to PL/I. When you specify this option, PL/I creates a dummy argument.

The NOMAP, NOMAPIN and NOMAPOUT options are effective only for structure arguments that are passed to COBOL. The options do not apply to scalars.

Use these options when program efficiency is important. These options help you avoid unnecessary mapping code.

Note: NOMAP, NOMAPIN, and NOMAPOUT can be specified on the same OPTIONS specification. Specifying NOMAPIN and NOMAPOUT for the same parameter is the same as specifying NOMAP for that parameter.

For information on interlanguage communication, see the *LE/VSE Programming Guide.*

ARGn ARG*n* refers to items in the argument list to which the NOMAP, NOMAPIN or NOMAPOUT option is to be applied. When you use ARG*n*, `ARG1` specifies that the option refers to the first argument, `ARG2` specifies that the option refers to the second argument, and so on. If you do not specify ARG*n*, the option is applied to all the data aggregates being passed.

ARG*n* values can appear in any order in the ARG*n* list. You should not repeat the same ARG*n* value in an ARG*n* list.

INTER is syntax checked and ignored. Refer to the *PL/I VSE Migration Guide* for more INTER information.

RETCODE

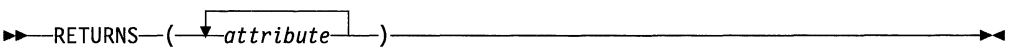
specifies that, on return from the non-PL/I routine, the fullword value in register 15 is to be saved as the PL/I return code. This option enables non-PL/I routines to pass return codes to PL/I. You can get the value of the return code using the PLIRETV built-in function.

An example using the OPTIONS attribute is:

```
DCL COBOLA OPTIONS(COBOL NOMAP(ARG1)
    NOMAPOUT(ARG3));
CALL COBOLA(X,Y,Z);                    /* X,Y,Z ARE STRUCTURES */

DCL ASSEM OPTIONS(ASM RETCODE)
    ENTRY(FIXED DEC,,,FLOAT);
CALL ASSEM(A,B,C,D);                   /* VALID   */
CALL ASSEM(A,B);                       /* VALID   */
CALL ASSEM;                            /* VALID   */
CALL ASSEM(A,,,D);                     /* INVALID */
```

RETURNS attribute

The RETURNS attribute specifies (within the invoking procedure) that the value returned from an external function procedure is *treated* as though it had the attributes given in the attribute list. The word treated is used because no conversion is performed in an invoking block upon any value returned to it. It further specifies, by implication, the ENTRY attribute for the name. Unless attributes for the returned value can be determined correctly by default, any invocation of an external function must appear within the scope of a declaration with the RETURNS attribute for the entry name. The syntax for the RETURNS attribute is:



If more than one attribute is specified, they must be separated by blanks (except for attributes such as precision, that are enclosed in parentheses).

The attributes that can be specified are any of the data attributes and alignment attributes for variables (except those for ENTRY variables), as shown in Table 12 on page 23. The OFFSET attribute can include an area reference.

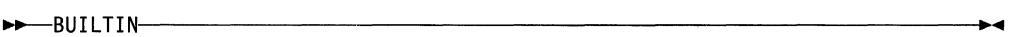
String lengths and area sizes must be specified by integers. The returned value has the specified length or size.

The RETURNS attribute must agree with the attributes specified in (or defaults for) the RETURNS option of the PROCEDURE or ENTRY statement to which the entry name is prefixed. The value returned will have attributes determined from the RETURNS option. If they do not agree, there is an error, since no conversion is performed.

If the RETURNS attribute is not specified for an external entry constant or an entry variable, the attributes for the value returned are set by default (as described in "Defaults for data attributes" on page 159).

BUILTIN attribute

The BUILTIN attribute specifies that a name is a built-in function name, pseudovariable name, or built-in subroutine name. The syntax for the BUILTIN attribute is:



Built-in names can be used as programmer-defined names, defined by explicit or implicit declaration. The BUILTIN attribute can be declared for a built-in name in any block that has inherited, from a containing block, some other declaration of the name. Consider the following examples:

Example 1

```
A: PROCEDURE;
      DECLARE SQRT FLOAT BINARY;
      X = SQRT;

      B:  BEGIN;
         DECLARE SQRT BUILTIN;
         Z = SQRT(P);
      END B;

   END A;
```

Example 2

```
A:  PROCEDURE;
SQRT: PROC(PARAM) RETURNS(FIXED(6,2));
      DECLARE PARAM FIXED (12);
      END SQRT;

   X = SQRT(Y);

   B:  BEGIN;
       DECLARE SQRT BUILTIN;
       Z = SQRT (P);
       END B;

   END A;
```

In both examples:

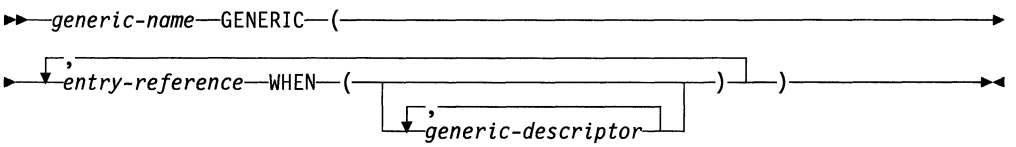
- In A, SQRT is a programmer-defined name.
- The assignment to the variable X is a reference to the programmer-defined name SQRT.
- In B, SQRT is declared with the BUILTIN attribute so that any reference to SQRT is recognized as a reference to the built-in function and not to the programmer-defined name SQRT declared in A.

In Example 2, if the procedure SQRT is an external procedure, procedure A needs the following statement to declare explicitly SQRT as an entry name, and to specify the attributes of the values passed to and returned from the programmer-written function procedure:

```
DCL SQRT ENTRY (FIXED (12))
         RETURNS (FIXED(6,2));
```

GENERIC attribute and references

A *generic name* is declared with the GENERIC attribute, and specifies a set of entry references. During compilation, other occurrences of the generic name are replaced by one of the set of entry references. Compilation then continues, using the entry reference. (Thus, generic declarations are a kind of a macro). Replacement is determined according to generic selection. The syntax for the GENERIC attribute is:



entry-reference

must not be based, 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, and can specify attributes that the corresponding argument must have in order that the associated entry reference can be selected for replacement. The following table shows the attributes that are allowed and their restrictions.

Attributes	Comments
ALIGNED	
AREA	No size can be specified
BINARY	
BIT	No length can be specified
CHARACTER	No length can be specified
COMPLEX	
DECIMAL	Dimension - No bounds may be specified; asterisks are used instead
ENTRY	No descriptor list can be specified
EVENT	
FILE	
FIXED	
FLOAT	
GRAPHIC	No length can be specified
LABEL	No label list can be specified
OFFSET	No area variable can be specified
PICTURE	*picture-specification*
POINTER	precision - number of digits and scaling factor must be specified
REAL	
UNALIGNED	
VARYING	

If a locator attribute (POINTER or OFFSET) is specified, the corresponding parameter must have the same attribute. No conversion from one type to the other can be performed when the entry-point is invoked.

Level numbers must not be specified. When an aggregate is passed as an argument to a generic entry name, no dummy argument is created.

Where no descriptor is required, it can be either omitted or indicated by an asterisk. The asterisk form is required if the missing descriptor is the only descriptor. For example, whereas (,) represents two descriptors, (*) represents one.

The generic-descriptor list which represents the absence of any argument takes the form:

```
....ENTRY1 WHEN( )...
```

Generic selection of a particular entry reference is based upon the arguments, or absence of all arguments, following the generic name. When a generic name is encountered, the number of arguments in the argument list following the generic name (if any), and attributes of each argument, are compared with each generic-descriptor list. The member that replaces the generic name is the first one whose generic-descriptor list matches the arguments both in number and attributes. The generic name is then replaced with the entry expression with the matching generic-descriptor list.

For example, if a generic-descriptor list contains:

```
(FLOAT,FIXED)
```

and the corresponding two arguments have attributes such as DECIMAL FLOAT(6) and BINARY FIXED(15,0) either declared or by default, each attribute in the generic-descriptor list is an attribute of the corresponding argument and the selection is successful. However, if either argument did not have the attributes in the corresponding descriptor, the selection process would consider the next generic member with just two descriptors. For example:

```
DECLARE CALC GENERIC
        (FXDCAL WHEN (FIXED,FIXED),
        FLOCAL WHEN (FLOAT,FLOAT),
        MIXED WHEN (FLOAT,FIXED));
        Z = X+CALC(X,Y);
```

The first statement defines CALC as a generic name having three members, FXDCAL, FLOCAL, and MIXED. One of these three entry references will be used to replace the generic name CALC, depending on the characteristics of the two arguments in that reference. If X and Y are floating-point and fixed-point, respectively, MIXED is the replacement.

In a similar manner, an entry point to a procedure can be selected by dimensionality. For example:

```
DCL  D GENERIC (D1 WHEN((*)),
                D2 WHEN((*,*))),
     A(2),
     B(3,5);
CALL D(A);
CALL D(B);
```

The generic name D in the first CALL statement is replaced by the entry expression D1. The generic name D in the second CALL statement is replaced by the entry expression D2.

If all the descriptors are omitted or consist of an asterisk, the first entry name 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.

An argument with the GENERIC attribute matches an ENTRY attribute in a generic descriptor list.

Generic names can be specified as arguments to nongeneric entry names.

If the nongeneric entry name is an entry variable or an external entry constant, it must be declared with a parameter descriptor list. The descriptor for the generic argument must be ENTRY with a parameter descriptor list. This nested list is used to select the argument to be passed. For example:

```
A:  PROC;
    DCL B GENERIC (C WHEN(FIXED),
                   D WHEN(FLOAT)),
        E ENTRY (ENTRY(FIXED));
    CALL E(B);
      .
      .
      .
    END A;
```

When procedure E is invoked, C is selected and passed as the argument, since the descriptor specifies that the parameter specified by the entry name parameter is FIXED.

If the nongeneric entry name is an internal entry constant, the corresponding parameter must be declared ENTRY with a parameter descriptor list. This list is used to select the argument to be passed. For example:

```
A:  PROC;
    DCL B GENERIC (C WHEN(FIXED),
                   D WHEN(FLOAT));
    CALL E(B);
    E:  PROC(P);
        DCL P ENTRY(FIXED);
          .
          .
          .
    END E;
END A;
```

When procedure E is invoked, C is selected and passed as the argument, since the parameter of entry name parameter is declared to be FIXED.

The program is in error if no generic-descriptor list is found to match the attributes of the arguments to a particular generic name.

Entry invocation or entry value

There are times when it may not be apparent whether an entry value itself is used or the value returned by the entry invocation is used.

First, if the entry reference has an argument list, even if null, it is always invoked. For example, `'E(1)'`.

All of the following are for the no-argument cases:

If the entry reference is used as an argument to a function that will not accept an argument of type ENTRY, the entry is invoked. For example:

```
DCL DATE BUILTIN;
Z = SUBSTR (DATE,5,2);
```

Date is invoked.

The function is not invoked when the entry reference is:

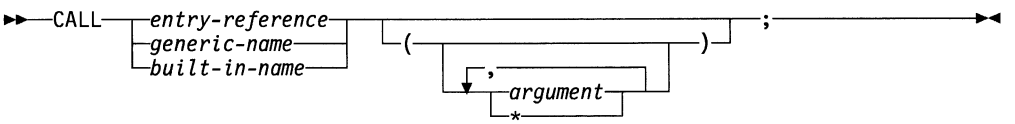
- Used as an argument to a function that will accept an ENTRY argument.
- The right-hand side of an assignment to an entry variable.
- In a comparison to another entry reference (this comparison can be implied by a SELECT statement).
- An argument passed to an entry parameter.
- Used in a context that requires an entry variable.
- Used as an argument to a generic name.
- An argument enclosed in parentheses. In the following example, the value of the entry B is passed as the argument to A:

  ```
  CALL A((B));
  ```

In all remaining cases, the entry is invoked.

CALL statement

The CALL statement invokes a procedure and transfers control to a specified entry point of the procedure. The syntax for the CALL statement is:



The options of the CALL statement can appear in any order.

entry-reference,
generic-name, or
built-in name
specifies the entry point of the subroutine to be invoked.

argument
Element, aggregate expression, or asterisk (*). PL/I has a limit of 64 arguments in a single CALL statement. See "Association of arguments and parameters" on page 128.

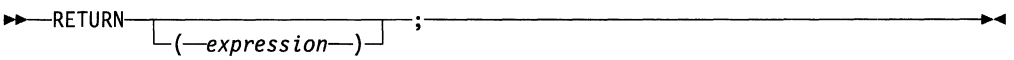
The environment of the invoked procedure is established after evaluation of any expressions and before the procedure is invoked.

A CALL statement must not be used to invoke a procedure if control is to be returned to the invoking procedure by means of a RETURN(expression) statement.

If the procedure invoked by the CALL statement has been specified in a FETCH or RELEASE statement, and if it is not present in main storage, the CALL statement initiates dynamic loading of the procedure from auxiliary storage. (See "Dynamic loading of an external procedure" on page 122.)

RETURN statement

The RETURN statement terminates execution of the procedure that contains the RETURN statement and returns control to the invoking procedure. The RETURN statement can also return a value. The syntax for the RETURN statement is:



The RETURN statement without *expression* terminates procedures invoked as subroutines; control is returned to the point following the CALL statement. If the RETURN statement terminates the main procedure, the FINISH condition is raised prior to program block termination.

The RETURN statement with *expression* is used to terminate a procedure invoked by a function reference. The value returned to the function reference is the value of the expression specified, converted to conform to the attributes for the invoked entry point. Control is returned to the function reference.

The compiler attempts to provide for the conversion of every RETURN expression to the RETURNS attributes of every entry point of the procedure. Some of these conversions may be invalid and may produce diagnostic messages when the procedure is compiled. At execution time, however, only the conversion applicable to the invoked entry point is performed.

Chapter 6. Data declaration

Chapter 6. Data declaration

This chapter discusses explicit and implicit declarations, scopes of declarations, multiple declarations, and defaults for data attributes.

When a PL/I program is executed, it can manipulate many different data items. Each data item, except an 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).

You can use the compiler ATTRIBUTES option to request a listing of names and their attributes.

Most attributes of data items are known at the time the program is compiled. For non-STATIC 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. Names with the attribute FILE can acquire additional attributes when the file is opened.

The DECLARE statement specifies some or all of the attributes of a name. Some attributes often are determined by context. If the attributes are not explicitly declared and cannot be determined by context, default attributes are applied. In some cases, the combination of defaults and context determination make it unnecessary to use a DECLARE statement.

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. This can make modification easier and the interpretation of compiler messages clearer.

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 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 PROCEDURE or BEGIN statement through the corresponding END statement (including condition prefixes of BEGIN and PROCEDURE statements), is said to be *contained in* that block. However, the

labels of the BEGIN or PROCEDURE statement heading the block, as well as the labels of any ENTRY statements that apply to the block, are not contained in that block. Nested blocks are contained in the block in which they appear.

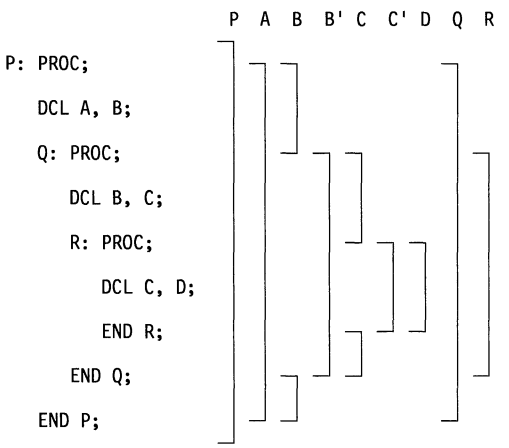
Text that is contained in a block, but not contained in any other block nested within it, is said to be *internal to* that block. Entry names of a procedure (and labels of a BEGIN statement) are not contained in that block. Consequently, they are internal to the containing block. Entry names of an external procedure are treated as if they were external to the external procedure.

Explicit declaration

A name is explicitly declared if it appears:

- In a DECLARE statement. The DECLARE statement explicitly declares attributes of names.
- In a parameter list. When a name appears in a parameter list, it is the same as if a DECLARE statement for that name appeared immediately following the PROCEDURE or ENTRY statement in which the parameter list occurs (though the same name can also appear in a DECLARE statement internal to the same block).
- As an entry constant. Labels of PROCEDURE and ENTRY statements constitute declarations of the entry constants within the containing procedure.
- As a label constant. A label constant on a statement constitutes an explicit declaration of the label.

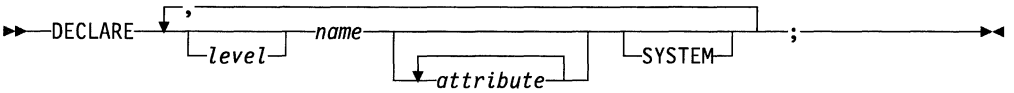
The scope of an explicit declaration of a name is that block to which the declaration is internal, including all contained blocks, except those blocks (and any blocks contained within them) to which another explicit declaration of the same name is internal. In the following example:



The lines to the right 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.

DECLARE statement

The DECLARE statement specifies attributes of a name and its position determines the scope of the declaration of the name. Other attributes for a name can be determined by default. Any number of names can be declared in one DECLARE statement. The syntax for the DECLARE statement is:



Abbreviation: DCL

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 names.

name Each *level 1* name must be unique within a particular block.

attribute

The attributes can appear in any order, except for the dimension and precision attributes. See Table 12 on page 23 and Table 13 on page 23, which show attributes classified according to data types.

All attributes given explicitly for the name must be declared together in a DECLARE statement, except that:

- Names having the FILE attribute can be given attributes in an OPEN statement (or have attributes implied by an implicit opening), as well. (See "OPEN statement" on page 244.)
- The parameter attribute is explicitly declared 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 (except that an INITIAL attribute given in one declaration in a compiled procedure need not be repeated).

The use of the OPTIONS attribute does not imply ENTRY.

SYSTEM

Specifies that the language-specified default attributes are to be applied to the name; attributes are not taken from DEFAULT statements. SYSTEM can appear before, after, or between the other attributes, but cannot immediately precede the dimension or precision attributes.

Computing dimension bounds, area sizes, and string lengths for automatic and DEFINED variables is done when the block that the declaration is internal to is activated. See "Block activation" on page 106.

Labels can be prefixed to DECLARE statements. A branch to such a label is treated as a branch to a null statement. Condition prefixes cannot be attached to a DECLARE statement.

Factoring of attributes

Attributes common to several names can be factored to eliminate repeated specification of the same attribute.

Factoring is achieved by enclosing the names in parentheses followed by the set of attributes which apply to all of the names. The dimension attribute can be factored. The precision attribute can be factored only in conjunction with an associated keyword attribute. Factoring can also be used on elementary names within structures. A factored level number must precede the parenthesized list.

Declarations 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. Factoring can be nested as shown in the fourth example below.

```
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.

Implicit declaration has the same effect as if the name were declared in the external procedure, even when all the occurrences of the name are internal to a block (called B, for example) that is contained in the external procedure. Consequently, the name is known throughout the entire external procedure, except for any blocks in which the name is explicitly declared. It is as if block B has inherited the declaration from the containing external procedure.

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 that appears in a CALL statement, in a CALL option, or is followed by an argument list is given the BUILTIN and INTERNAL attributes.
- 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 an EVENT option or in a WAIT statement is given the EVENT 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.

Since 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. For example, the following procedure is invalid:

```
P:  PROC (F);
    READ FILE(F) INTO(X);
    END P;
```

The name F is in a parameter list and is, therefore, explicitly declared. The language-specified default attributes REAL DECIMAL FLOAT conflict with the attributes that would normally be given to F by its appearance in the FILE option.

Scopes of declarations

Figure 8 illustrates the 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'.

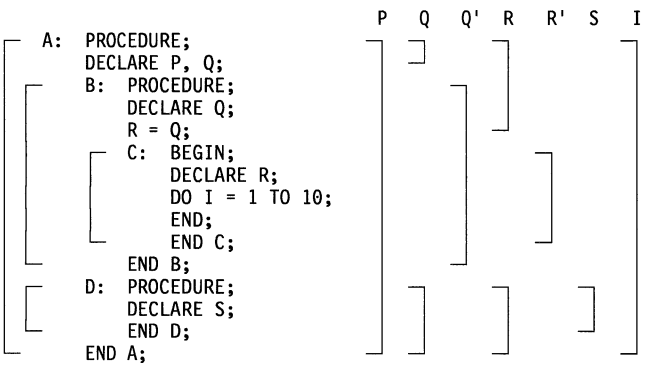


Figure 8. Scopes of data declarations

P is declared in the block A and known throughout A since it is not redeclared.

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.

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 8) with different scopes exist. The scope of the explicitly declared R is block C; the scope of the implicitly declared R is all of A except block C.

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.

S is explicitly declared in procedure D and is known only within D.

Figure 9 illustrates the scopes of entry constant and statement label declarations. The example shows two external procedures. 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. The scope of the declaration of the name A is only all of the block A, and not E.

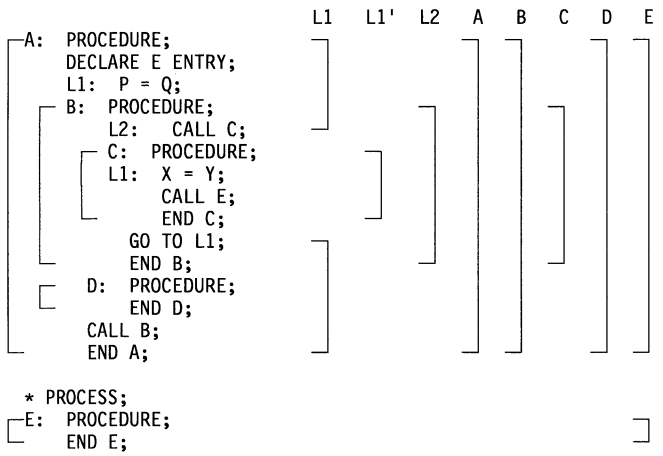


Figure 9. Scopes of entry and label declarations

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.

D and B are explicitly declared in block A and can be referred to anywhere within A; but since they are INTERNAL, they cannot be referred to in block E.

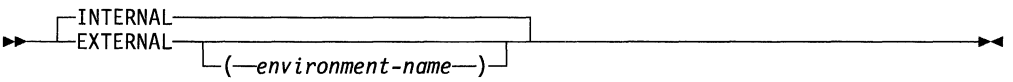
C is explicitly declared in B and can be referred to from within B, but not from outside B.

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

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, nonoverlapping scope.

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 program. The syntax for the INTERNAL and EXTERNAL attributes is:



Abbreviations: INT for INTERNAL
EXT for EXTERNAL

environment-name
specifies the name by which an assembler entry 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.

You can only specify the environment name for OPTIONS(ASSEMBLER) entry constants. You cannot specify the environment name in the DEFAULT statement. PL/I does not monocase the name that you specify. The name must:

- Be up to 8 characters in length.
- Be a character constant that does not contain a repetition factor, hex character constants, or mixed character constants.
- Differ from names internally generated by the compiler. Using 7-character names or shorter names ensures this.
- Be a valid name for the operating system.

In the following example, entry XYZ is declared with an environment name of ABC. The environment name of ABC is generated in the compiled object module rather than the language entry name of XYZ. The call to XYZ results in the invocation of ABC.

```
Dcl XYZ entry external('ABC') options(asm);
Call XYZ;               /* ABC is invoked            */
```

In the following example, the FETCH and CALL statements for entry ET resolve to entry FF during execution; FF is dynamically loaded and called.

```
Dcl ET entry external('FF') options(asm);
Fetch ET;               /* Operating system loads FF */
Call ET;                /* FF is invoked             */
```

INTERNAL is the default for entry names of internal procedures and for automatic, static, and based variables.

EXTERNAL is the default for controlled variables, file constants, entry constants, and programmer-defined conditions.

When a major structure name is declared EXTERNAL in more than one block, the attributes of the structure members must be the same in each case, although the corresponding member names need not be identical. For 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 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. It might be impossible for the compiler to check all declarations, particularly if the names are declared in different external procedures, so care should be taken to ensure that different declarations of the same name with the EXTERNAL attribute have matching attributes. You can use the attribute listing, which is available as optional output, to check the attributes of names. The following example illustrates the above points in a program:

```
A: PROCEDURE;
   DECLARE S CHARACTER (20);
   DCL SET ENTRY(FIXED DECIMAL(1)),
   OUT ENTRY(LABEL);
   CALL SET (3);
E: GET LIST (S,M,N);
B: BEGIN;
     DECLARE X(M,N), Y(M);
     GET LIST (X,Y);
     CALL C(X,Y);

C:   PROCEDURE (P,Q);
       DECLARE
         P(*,*),
         Q(*),
         S BINARY FIXED EXTERNAL;
       S = 0;
       DO I = 1 TO M;
         IF SUM (P(I,*)) = Q(I)
            THEN GO TO B;
         S = S+1;
         IF S = 3
            THEN CALL OUT (E);
         CALL D(I);
 B:    END;
     END C;
```

```
D:  PROCEDURE (N);
      PUT LIST ('ERROR IN ROW ',
            N, 'TABLE NAME ', S);
     END D;
   END B;

   GO TO E;
   END A;
* PROCESS;
OUT:  PROCEDURE (R);
      DECLARE R LABEL,
        (M,L) STATIC INTERNAL
          INITIAL (0),
        S BINARY FIXED EXTERNAL,
        Z FIXED DECIMAL(1);
      M = M+1; S=0;
      IF M<L
        THEN STOP;
        ELSE GO TO R;
SET:  ENTRY (Z);
      L=Z;
      RETURN;
      END OUT;
```

A is an external procedure name; its scope is all of block A, plus any other blocks where A is declared as external.

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.

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 (15,0), BINARY, and INTERNAL.

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.

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 an explicit declaration. However, a separate DECLARE statement is required in order to specify that P and Q are arrays; this is the explicit declaration. 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).

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.

The second external procedure in the example has two entry names, SET and OUT. The use of the names as entry constants constitutes an explicit declaration with the ENTRY and EXTERNAL attributes. They also must be declared explicitly with the ENTRY attribute in procedure A. Since ENTRY implies EXTERNAL, the two entry constants SET and OUT are known throughout the two external procedures.

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.

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, since 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 A is passed as an argument to the label parameter R.

The statement GO TO R transfers control to the label E, even though E is declared within A, and not known within OUT.

The variables M and L are declared as STATIC within the OUT procedure-block; their values are preserved between calls to OUT.

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.

Multiple declarations

Two or more declarations of the same name constitute a multiple declaration when any of the duplicate names are:

- Declared as internal to the same block, except when structure qualification makes references unique
- In a program of the same EXTERNAL name, but with different attributes

Defaults for data attributes

Every name in a PL/I source 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 *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.

The keyword SYSTEM can be specified in the DECLARE statement for a name to specify that language-specified defaults are to be applied and that attributes are not to be taken from DEFAULT statements.

Language-specified defaults

When a problem-data name has not been declared with a data type or when the RETURNS option is omitted from a function procedure, the default is coded arithmetic problem data.

If mode, scale, **and** base are not specified by a DECLARE or DEFAULT statement, or by a RETURNS option, variables with names beginning with any of the letters I through N are given the attributes REAL FIXED BINARY (15,0), and those with names beginning with any other alphabetic character or with a non-EBCDIC DBCS character are given the attributes REAL FLOAT DECIMAL (6).

A scaling factor in the precision attribute constitutes an explicit declaration of FIXED.

If mode, string, **or** base is specified by a DECLARE or DEFAULT statement, or by a RETURNS option, the remaining attributes are completed from the following list of defaults:

The default base is DECIMAL
The default scale is FLOAT
The default mode is REAL

Default precisions are then completed from the following list:

(5,0) for DECIMAL FIXED
(15,0) for BINARY FIXED
(6) for DECIMAL FLOAT
(21) for BINARY FLOAT

For example the statement:

```
DCL I BINARY(15)     /* no scale specified */
```

gets the attributes REAL and FLOAT. Whereas, the following statement:

```
DCL I BINARY(15,0)   /* scale specified    */
```

gets the attributes REAL and FIXED.

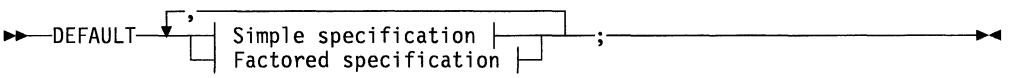
The language-specified defaults for scope, storage, and alignment attributes are shown in Table 12 on page 23 and Table 13 on page 23.

If no parameter descriptor list is given, the default is that the argument attributes match the parameter attributes.

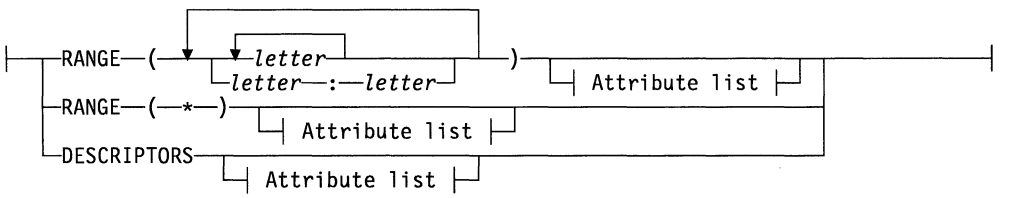
DEFAULT statement

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.

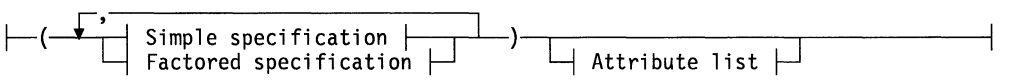
Structure elements are given default attributes according to the name of the element, not the qualified structure element name. The DEFAULT statement cannot be used to create a structure. The syntax for the DEFAULT statement is:



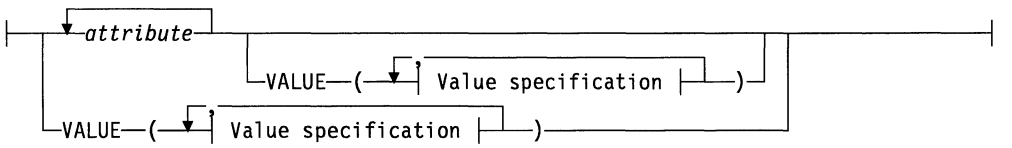
Simple specification:



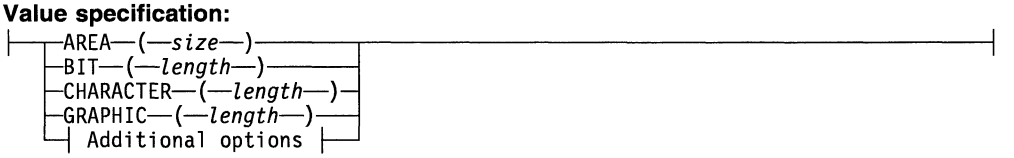
Factored specification:



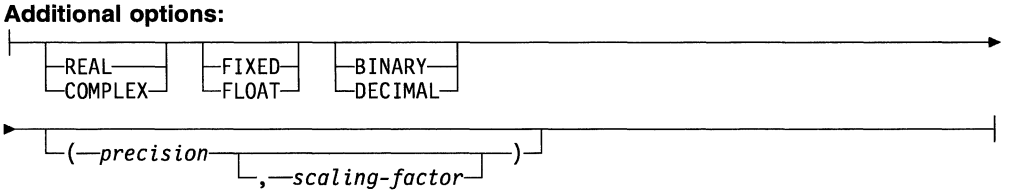
Attribute list:



Value specification:



Additional options:



Abbreviation: DFT

RANGE(letter)

specifies that the defaults apply to names that begin with the name(s) specified. Letter can be any letter in the English alphabet. For example:

```
RANGE (ABC)
```

applies to these names:

```
ABC
ABCD
ABCDE
```

but not to:

```
ABD
ACB
AB
A
```

Hence a single letter in the range-specification applies to all names that start with that letter. The RANGE letter can also be a non-EBCDIC DBCS character.

RANGE(letter:letter)
specifies that the defaults apply to names with initial letters that either correspond to the two letters specified, or to any letters between the two in alphabetic sequence. The letters cannot be DBCS. The letters given in the specification must be in increasing alphabetic order. For example:

```
RANGE(A:G,I:M,T:Z)
```

RANGE(*)
specifies all names in the scope of the DEFAULT statement. For example:

```
DFT RANGE (*) PIC '99999';
```

This statement specifies default attributes REAL PICTURE '99999' for all names.

An example of a *factored-specification* with the range options is:

```
DEFAULT (RANGE(A)FIXED, RANGE(B)
        FLOAT)BINARY;
```

This statement specifies default attributes FIXED BINARY for names with the initial letter A, and FLOAT BINARY for those with the initial letter B.

DESCRIPTORS
specifies that the attributes are included in any parameter descriptors in a parameter descriptor list of an explicit entry declaration, provided that:

- The inclusion of any such attributes is not prohibited by the presence of alternative attributes of the same class.
- At least one attribute is already present. (The DESCRIPTORS default attributes are not applied to null descriptors.)

For example:

```
DEFAULT DESCRIPTORS BINARY;
DCL X ENTRY (FIXED, FLOAT);
```

The attribute BINARY is added to each parameter descriptor in the list, producing the equivalent list:

```
(FIXED BINARY, FLOAT BINARY)
```

attribute-list
specifies a list of attributes from which selected attributes are applied to names in the specified range. 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.

The file description attributes, and the attributes ENTRY, RETURNS, LIKE, and VARIABLE cannot be used in an attribute-list. If FILE is used, it implies the attributes VARIABLE and INTERNAL.

These are the attributes and their restrictions:

AREA without a size specification

BIT, CHARACTER, or GRAPHIC
: without a string length specification

LABEL without a label list

Arithmetic base, mode, and scale attributes
: without precision specifications

CONTROLLED
: for a parameter name, a specification of CONTROLLED as a default attribute is ignored. The CONTROLLED attribute is also ignored if it appears in a DESCRIPTORS attribute specification.

dimension
: The dimension attribute is allowed, but only as the first item in an attribute specification. The bounds can be specified as an arithmetic constant or an expression and can include the REFER option. For example:

```
DFT RANGE(J) (5);
DFT RANGE(J) (5,5) FIXED;
```

but not

```
DFT RANGE(J) FIXED (5);
```

Although the DEFAULT statement can specify the dimension attribute for names that have not been declared explicitly, a subscripted name is contextually declared with the attribute BUILTIN. Therefore, the dimension attribute can be applied by default only to explicitly declared names.

The INITIAL attribute can be specified.

Attributes that conflict, when applied to a data item, do not necessarily conflict when they appear in an attribute specification. For example:

```
DEFAULT RANGE(S) BINARY VARYING;
```

This means that any name that begins with the letter S and is declared explicitly with the BIT or CHARACTER attribute receives the VARYING attribute; all others (that are not declared explicitly or contextually as other than arithmetic data) receive the BINARY attribute.

VALUE can appear anywhere within an attribute-specification except before a dimension attribute.

VALUE establishes any default rules for an area size, string length, and precision.

The size of AREA data, or length of BIT, CHARACTER, or GRAPHIC data, can be an expression or an integer and can include the REFER option, or can be specified as an asterisk.

For example:

```
DEFAULT RANGE(A:C)
        VALUE (FIXED DEC(10),
               FLOAT DEC(14),
               AREA(2000));
DECLARE B FIXED DECIMAL,
        C FLOAT DECIMAL,
        A AREA;
```

These statements are equivalent to:

```
DECLARE B FIXED DECIMAL(10),
        C FLOAT DECIMAL(14),
        A AREA(2000);
```

The base and scale attributes in value-specification must be present to identify a precision specification with a particular attribute. The base and scale attributes can be factored (see "Factoring of attributes" on page 153).

The only attributes that the VALUE option can influence are area size, string length, and precision. Other attributes in the option, such as CHARACTER and FIXED BINARY in the above examples, merely indicate which attributes the value is to be associated with. Consider the following example:

```
DEFAULT RANGE(I) VALUE(FIXED DECIMAL(8,3));
I = 1;
```

If it is not declared explicitly, I is given the language-specified default attributes FIXED BINARY(15,0). It will *not* be influenced by the default statement, because this statement specifies only that the default precision for FIXED DECIMAL names is to be (8,3).

For example:

```
DFT RANGE(*) VALUE(FIXED BINARY(31));
```

specifies precision for identifiers already known to be FIXED BINARY, while

```
DFT RANGE(*) FIXED BINARY VALUE(FIXED BINARY(31));
```

specifies both the FIXED BINARY attribute as a default and the precision.

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;
L1:  DEFAULT RANGE (XY) FIXED;
 Q:  BEGIN;
L2:  DEFAULT RANGE (XYZ) FLOAT;
     END P;
```

The scope of DEFAULT statement L1 is procedure P and the contained block Q. The range of DEFAULT statement L1 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.

Labels can be prefixed to DEFAULT statements. A branch to such a label is treated as a branch to a null statement. Condition prefixes cannot be attached to a DEFAULT statement.

Programmer-defined default for the RETURNS option

The default attributes of values returned from function procedures are dependent on the entry name used to invoke the procedure. The DEFAULT statement can be used to specify these attributes when the entry name, or the initial letter of the entry name, is specified in the DEFAULT statement.

For example:

```
DEFAULT RANGE (X) FIXED BINARY;
X :  PROC(Y);
```

would be interpreted as:

```
X :  PROC(Y) RETURNS (FIXED BINARY);
```

Restoring language-specified defaults

The following statement:

```
DEFAULT RANGE(*), DESCRIPTORS;
```

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.

Chapter 7. Statements

Chapter 7. Statements

This chapter lists all PL/I statements. If they are described in other chapters, a pointer to the relevant chapter is given.

Statements that direct the operation of the compiler, rather than contributing to the program produced by the compiler, begin with a percent symbol (%) or an asterisk (*). This chapter discusses these % statements that allow you to control the source program listing and to include external strings in the source program. Preprocessor statements can also begin with a %.

Listing control statements are %PRINT, %NOPRINT, %PAGE, and %SKIP. These statements cannot be a *unit* of a compound statement.

%ACTIVATE statement

The %ACTIVATE statement is described in "%ACTIVATE statement" on page 422.

ALLOCATE statement

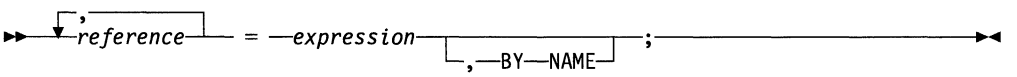
The ALLOCATE statement is described in "ALLOCATE statement for controlled variables" on page 204 and "ALLOCATE statement for based variables" on page 214.

Assignment statement

The assignment statement evaluates an expression and assigns its value to one or more target variables. The target variables can be element, array, or structure variables, or pseudovariables.

The assignment 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 made. See "GET statement" on page 269 for information about the GET statement and "PUT statement" on page 269 for information about the PUT statement.)

Since the attributes of the variable or pseudovariable on the left can differ from the attributes of the result of the expression (or of the variable or constant), the assignment statement might require conversions (see Chapter 4, "Data conversion"). The syntax for the assignment statement is:



Area assignment is described in "Area assignment" on page 221.

An **element assignment** is 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.

For **array assignments,** each target variable must be an array. The right-hand side can be a structure, array, or element expression. If the right-hand side contains arrays of structures, all target variables must be arrays of structures. The BY NAME option can be given only when the right-hand side contains at least one structure.

For **structure assignments,** each target variable must be a structure. The right-hand side can be a structure or element expression.

Aggregate assignments (array and structure assignments) are expanded into a series of element assignments as follows. The label prefix of the original statement is applied to a null statement preceding the other generated statements. Array and structure assignments, when there are more than one, are done iteratively. 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 could 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.

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 (for instance, if the arrays are not interleaved, or if the structures are contiguous and have the same format).

Because of the many possible variations in structuring, some mismatches with the elements might not be detected when structure assignments are done. This is usually true when the structures in the aggregate assignment contain arrays.

In **array assignments,** conceptually, 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 original assignment statement has a BY NAME option, the generated assignment statement is given a BY NAME option. If the generated assignment statement is a structure assignment, it is expanded as given below.

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 steps 1 through 3 below. Steps 1 through 3 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: 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. If the generated assignment is a structure or array-of-structures assignment, BY NAME is appended. All generated assignment statements are given the condition prefix of the original assignment statement.
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

The values of the expression in an assignment statement can be assigned to more than one variable or pseudovariable. 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;
```

If multiple assignment is used for a structure assignment BY NAME, the elementary names affected are only those that are common to all of the structures referenced in the statement.

Examples of assignment statements

The following example of the assignment statement can be used for internal data movement. The value of the expression to the right of the assignment symbol is to be assigned to the variable to the left.

```
NTOT=TOT;
```

The following example includes an expression whose value is to be assigned to the variable to the left of the assignment symbol:

```
AV=(AV*NUM+TAV*TNUM)/(NUM+TNUM);
```

The following two examples illustrate 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;

ONE = TWO, BY NAME;
ONE.PART1 = THREE.PART1, BY NAME;
ONE = TWO + THREE, BY NAME;
```

The first assignment statement is the same as the following:

```
ONE.PART1.RED    = TWO.PART1.RED;
ONE.PART2.YELLOW = TWO.PART2.YELLOW;
```

The second assignment statement is the same as the following:

```
ONE.PART1.RED = THREE.PART1.RED;
```

The third assignment statement is the same as the following:

```
ONE.PART1.RED = TWO.PART1.RED
              + THREE.PART1.RED;

ONE.PART2.YELLOW = TWO.PART2.YELLOW
              + THREE.PART2.YELLOW;
```

%assignment statement

The %assignment statement is described in "%assignment statement" on page 422.

BEGIN statement

The BEGIN statement is described in "BEGIN statement" on page 132.

CALL statement

The CALL statement is described in "CALL statement" on page 147.

CLOSE statement

The CLOSE statement is described in "CLOSE statement" on page 249.

%DEACTIVATE statement

The %DEACTIVATE statement is described in "%DEACTIVATE statement" on page 423.

DECLARE statement

The DECLARE statement is described in "DECLARE statement" on page 152.

%DECLARE statement

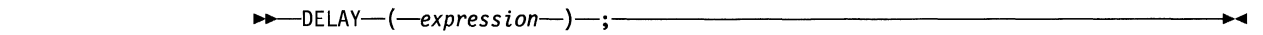
The %DECLARE statement is described in "%DECLARE statement" on page 423.

DEFAULT statement

The DEFAULT statement is described in "DEFAULT statement" on page 161.

DELAY statement

The DELAY statement suspends execution for at least a specified period of time. The maximum wait time is 7 hours 46 minutes. The syntax for the DELAY statement is:



expression
Evaluated and converted to a fixed-point binary value of precision (31,0). Execution is suspended for the number of milliseconds specified by *expression*.

For example, this statement suspends execution for 20 milliseconds:

```
DELAY (20);
```

DELETE statement

The DELETE statement is described in "DELETE statement" on page 255.

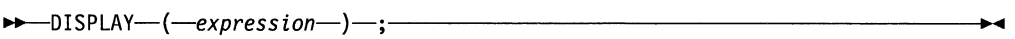
DISPLAY statement

The DISPLAY statement displays a message on the user's terminal or on the system console. A response might be requested from the operator. DISPLAY also can be used with the REPLY option to allow operator communication with the program by typing in a code or message. The REPLY option will suspend program execution until the operator acknowledges the message. The character data for DISPLAY or REPLY can contain mixed character data.

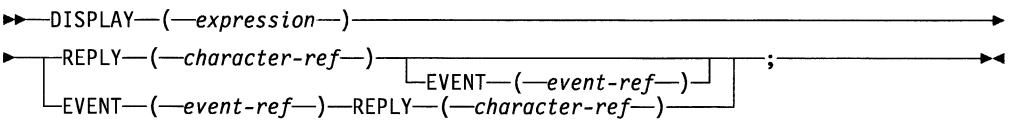
To display GRAPHIC data, you must use CHAR BUILTIN to convert the GRAPHIC data to mixed-character data.

If GRAPHIC data was entered in the REPLY, it is received as character data that contains mixed data. This can then be converted to GRAPHIC data using the GRAPHIC BUILTIN. The syntax for the DISPLAY statement is:

Type 1:



Type 2:



expression
Converted, where necessary, to a varying character string. This character string is the message displayed. It can contain mixed character data.

REPLY Receives a string that is an operator-supplied message. The STRING pseudovariable must not be used. The reply message can contain CHARACTER, GRAPHIC, or mixed data. If the reply contains GRAPHIC data it is assigned, along with the shift control characters, to the character string reference. The content of the reply is not checked for matched pairs of shift control characters.

EVENT If the EVENT option is not specified, execution is suspended until the operator's message is received.

If the EVENT option is given, execution does not wait for the reply to be completed before continuing with subsequent statements. The status of the event variable is set to 0, and the completion part of the event-variable is given the value '0'B until the reply is completed, when it is given the value '1'B. The reply is complete only after the execution of a WAIT statement naming the event. Another DISPLAY statement must not be executed until the previous reply is complete.

REPLY and EVENT can appear in any order.

Example of the DISPLAY statement

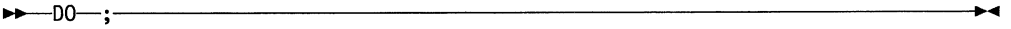
The following statement displays the message END OF JOB.

```
DISPLAY ('END OF JOB');
```

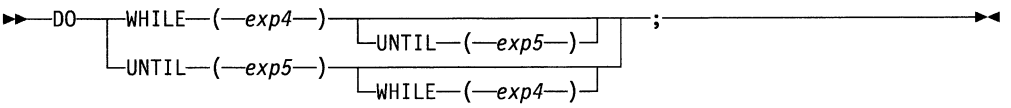
DO statement

The DO statement and its corresponding END statement delimit a group of statements collectively called a do-group. The DO statement can specify conditions for repeatedly running the do-group. The syntax for the DO statement is:

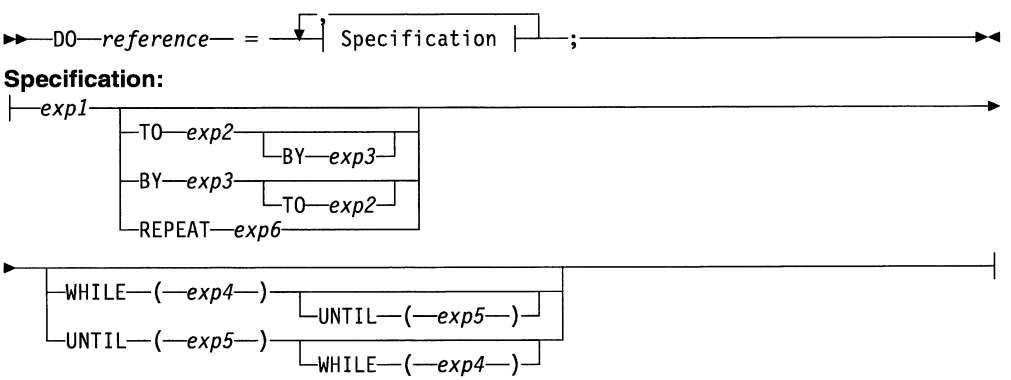
Type 1:



Type 2:



Type 3:



Note: Expression*n* is abbreviated as exp*n*.

WHILE exp4

Before each do-group repetition, 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:

- *Type 2 do-groups:* the do-group ends. Control passes to the statement following the do-group.
- *Type 3 do-groups:* the *specification* containing this WHILE option ends. Control passes to the next *specification* (if one exists) or the statement following the do-group.

UNTIL exp5

After each do-group repetition, exp5 is evaluated and, if necessary, converted to a bit string.

If all bits in the resulting string are 0, or the string is null, the do-group is executed.

If any bit is 1:

- *Type 2 do-groups:* the do-group ends. Control passes to the statement following the do-group.
- *Type 3 do-groups:* the *specification* containing this UNTIL option ends. Control passes to the next *specification* (if one exists) or the statement following the do-group.

reference

The *control variable*. Execution of a do-group *specification* ends when the control variable, tested at the end of each repetition, is out of range. Control passes to the next *specification* (if one exists) or the statement following the do-group END statement. The control variable is out of range when:

- The BY value is positive and the control variable is > the TO value
- The BY value is negative and the control variable is < the TO value

If *reference* is a program-control variable, the BY and TO options cannot be used in *specification*. (The following pseudovariables cannot be used: COMPLETION, COMPLEX, and STRING.) All data types are allowed.

If you refer to a control variable after its do-group has ended, the control variable has the out of range value that caused the do-group to end.

BASED or CONTROLLED control variables only:

The generation, g, of a control variable is established once at the start of the do-group, immediately before the initial value expression (exp1) is evaluated. If the control variable generation is changed in the do-group (to h) the do-group continues to execute with the control variable derived from the generation g. However, any reference to the control variable inside the do-group is a reference to generation h. It is an error to free generation g in the do-group.

For example, the FREE statement at LABEL is in error:

```
     DECLARE I CONTROLLED;
     ALLOCATE I;
     DO I = 1 TO 10; /* This is generation G of I                    */
         ALLOCATE I; /* This is generation H of I                    */
         .
         .
         .
         I = 20;     /* Generation H of I is assigned a value of 20  */
         J = I;      /* J is assigned the value of generation H of I */
                     /* that is, J = 20                              */
         FREE I;     /* Free generation H of I                       */
LABEL:   FREE I;     /* THIS IS AN ERROR!                            */
                     /* You cannot free generation G of I            */
         .
         .
         .
     END;
```

exp1 The initial value of the control variable.

If you omit *TO exp2*, *BY exp3*, and *REPEAT exp6* from a *specification*, the do-group executes once only, with a control variable value of *exp1*. If you include *WHILE(exp4)*, the do-group does not execute unless *exp4* is true.

TO exp2

The terminating value of the control variable. *exp2* is evaluated at entry to the specification and saved. Execution of a do-group *specification* ends when the control variable, tested at the end of each repetition, is out of range. When this happens, control passes to the next *specification* (if one exists) or the statement following the do-group.

If you omit *TO exp2* and specify *BY exp3*, the do-group repeats indefinitely, unless it is terminated by a WHILE or UNTIL option, or a statement transfers control out of the do-group.

BY exp3

The increment added to the control variable after each do-group repetition. *exp3* is evaluated at entry to the specification and saved.

If you omit *BY exp3* and specify *TO exp2*, *exp3* defaults to 1.

If you specify BY 0, the do-group repeats indefinitely unless it is halted by a WHILE or UNTIL option, or a statement transfers control out of the do-group.

REPEAT

exp6 is evaluated and assigned to the control variable after each do-group repetition. The do-group repeats until it is terminated by the WHILE or UNTIL option, or a statement transfers control out of the do-group.

Type 1 specifies that the statements in the group are executed; it does not provide for the repetitive execution of the statements within the group.

Do-group types 2 and 3 provide for the repetitive execution of the statements within the do-group. The TO and BY options let you vary the control variable 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).

The effect of executing a do-group with one specification can be summarized as follows:

1. If *control variable* 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 control variable. Then the initial value is assigned to *control variable*. For example:

   ```
   DO control-variable = 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 in which *p* is a compiler-created pointer; *v* is a compiler-created based variable based on *p* and with the same attributes as the control variable; and *e1*, *e2*, and *e3* are compiler-created variables:

```
p=ADDR(variable);
e1=exp1;
e2=exp2;
e3=exp3;
v=e1;
```

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 control variable:
   a. If the TO or BY option is specified, add the previously-evaluated exp3 (e3) to the control variable.
   b. If the REPEAT option is specified, evaluate the exp6 and assign it to the control variable.
   c. If the TO, BY, and REPEAT options are all absent, leave the do-group.
7. Go to 2.

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.

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. Consequently, Type 2 and 3 do-groups cannot contain ENTRY statements. Control can also return to a do-group from a procedure or ON-unit invoked from within that do-group.

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 statement. 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 TO and BY: 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
```

For a variable that is not a pseudovariable, the above action of the do-group definition is equivalent to the following expansion. In this expansion, *p* is a compiler-created pointer; *v* is a compiler-created based variable based on a *p* and with the same attributes as *variable*; and *e1*, *e2*, and *e3* are compiler-created variables:

```
LABEL:  p=ADDR(variable);
        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
```

For a variable that is not a pseudovariable, the above action of the do-group definition is equivalent to the following extension:

```
LABEL:  p=ADDR(variable);
        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 above expansions follow:

1. The above 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.

Examples of DO statements

The DO statement can specify a group of statements to be executed in the THEN clause or the ELSE clause of an IF statement, or in the WHEN statement or the OTHERWISE statement in a select-group. For example:

```
IF A=B
   THEN DO;
        .
        .
        .
        END;
   ELSE DO I=1 TO 2;
        .
        .
        .
        END;
```

A repetitive do-group might take the form:

```
DO I = 1 TO 10;
  .
  .
  .
END;
```

In this example, the do-group is executed ten times, while the value of the control variable I ranges from 1 through 10. The effect of the DO and END statements is equivalent to the following:

```
   I = 1;
A: IF I > 10 THEN GO TO B;
     .
     .
     .
   I = I +1;
   GO TO A;
B: next statement
```

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 control variable are required, the BY option must be used. For example:

```
DO I = 10 TO 1 BY -1;
```

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;
```

The preceding example is equivalent to the following:

```
    I=1;
A:  .
    .
    .
    I=2*I;
    GOTO A;
```

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;
```

The WHILE and UNTIL options make successive executions of the do-group dependent upon a specified condition. For example:

```
DO WHILE (A=B);
    .
    .
    .
END;
DO UNTIL (A=B);
    .
    .
    .
END;
```

The DO WHILE statement is equivalent to the following:

```
S:  IF A=B THEN;
        ELSE GOTO R;
    .
    .
    .
    GOTO S;
R:  next statement
```

The DO UNTIL statement 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.

If, in the following example, when the DO statement is first encountered, A¬=B, the do-group is not executed at all. If, however, A=B, the do-group is executed. If, after an execution of the do-group, X=10, no further executions are performed. Otherwise, a further execution is performed provided that A is still equal to B.

```
DO WHILE(A=B) UNTIL(X=10);
```

In the following example, the do-group is executed at least once, with I equal to 1. If, after an execution of the do-group, Y=1, 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.

```
DO I=1 TO 10 UNTIL(Y=1);
```

In the following example, the first execution of the do-group is performed with I=1. 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.

```
DO I=1 REPEAT 2*I UNTIL(I=256);
```

The following example shows a DO statement used to step along a chained list. 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.

```
DO P=PHEAD REPEAT P -> FWD
      WHILE(P¬=NULL());
```

The following DO statement specifies that the do-group executes once with the value of NAME set equal to the value of 'TOM', once with the value of NAME set equal to the value of 'DICK', and once with the value of NAME set equal to the value of 'HARRY'.

```
DO NAME = 'TOM','DICK','HARRY';
```

The following statement specifies that the do-group executes a total of 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;
```

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);
```

The following statement specifies that the do-group is to be executed nine times, with the value of I equal to 1 through 9, and 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);
```

The control variable 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. For example:

```
DO I = 1 TO 10;
   A(I) = I;
   END;
```

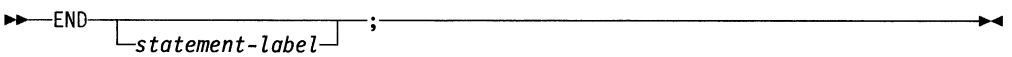
In this example, the first ten elements of A are set to 1,2,...,10, respectively.

%DO statement

The %DO statement is described in "%DO statement" on page 424.

END statement

The END statement ends one or more blocks or groups. 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, BEGIN, or PROCEDURE statement having that statement-label. It also closes any unclosed groups or blocks physically within that group or block; this is known as *multiple closure*.

If a statement-label does not follow END, the END statement closes the one group or block headed by the nearest preceding DO, SELECT, 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 "Block termination" on page 107 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.

Multiple closure

Multiple closure is a shorthand method of specifying a number of consecutive END statements. For example:

```
FRST: PROCEDURE;
      statement-f1
      statement-f2
      ABLK: BEGIN;
            statement-a1
            statement-a2
            SCND: PROCEDURE;
                  statement-s1
                  statement-s2
                  BBLK: BEGIN;
                        statement-b1
                        statement-b2
                        END;
                  END;
            statement-a3
            END ABLK;
      END FRST;
```

In this example, there are no statements between the END statements for begin block BBLK and internal procedure SCND. This is also true for begin block ABLK and external procedure FRST. In such cases, it is not necessary to use an END statement for each block, as shown; rather, one END statement can be used to end BBLK and SCND, and another END can be used to end ABLK and FRST. In the first case, the statement would be END SCND, because one END statement with no following label would close only the begin block BBLK. In the second case, only the statement END FRST is required; Thus, the example can be specified as follows:

```
FRST: PROCEDURE;
      statement-f1
      statement-f2
      ABLK: BEGIN;
            statement-a1
            statement-a2
            SCND: PROCEDURE;
                  statement-s1
                  statement-s2
                  BBLK: BEGIN;
                        statement-b1
                        statement-b2
                  END SCND;
            statement-a3
      END FRST;
```

All intervening groups and blocks are terminated if control passes to an END statement specifying multiple closure. For example:

```
CBLK:    PROCEDURE;
         statement-c1
         statement-c2
DGP:     DO I = 1 TO 10;
         statement-d1
         GO TO LBL;
         statement-d2
LBL:     END CBLK;
```

In this example, the END CBLK statement closes the block CBLK and the repetitive do-group DGP. The effect is as if an unlabeled END statement for DGP appeared immediately after statement-d2, so that the transfer to LBL would prevent all but the first execution of DGP from taking place.

%END statement

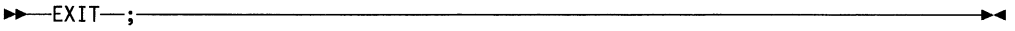
The %END statement is described in "%END statement" on page 425.

ENTRY statement

The ENTRY statement is described in "ENTRY statement" on page 111.

EXIT statement

The EXIT statement immediately terminates the program. The syntax for the EXIT statement is:



Prior to any termination activity, the FINISH condition is raised. On normal return from the FINISH ON-unit, the program terminates.

FETCH statement

The FETCH statement is described in "FETCH statement" on page 123.

FORMAT statement

The FORMAT statement is described in "FORMAT statement" on page 270.

FREE statement

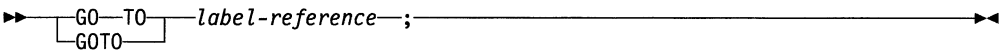
The FREE statement is described in "FREE statement for controlled variables" on page 206 and "FREE statement for based variables" on page 215.

GET statement

The GET statement is described in "GET statement" on page 269.

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:



label-reference

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 119).

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 121).

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.

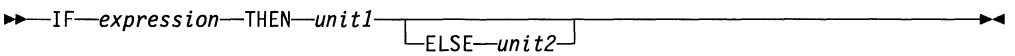
If the destination of the GO TO is specified by a label variable, it can then be used as a switch by assigning label constants to the label variable. If the label variable is subscripted, the switch can be controlled by varying the subscript. By using label variables or function references, quite complex switching can be effected. It is usually true, however, that simple control statements are the most efficient.

%GO TO statement

The %GO TO statement is described in "%GO TO statement" on page 425.

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. The syntax for the IF statement is:



expression
Evaluated and, if necessary, converted to a bit string.

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, ENTRY, FORMAT, PROCEDURE, or a % statement. If you use one of these statements, 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; *unit1*, *unit2*, or both can be an IF statement. 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.

Examples of IF statements

In the following example, if the comparison is true (A is equal to B), the value of D is assigned to C, and the ELSE unit is not executed. 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.

```
IF A = B THEN
  C = D;
ELSE
  C = E;
```

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 statement is described in "%IF statement" on page 425.

%INCLUDE statement

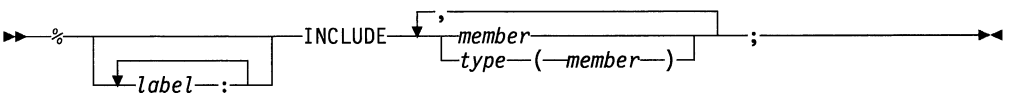
The %INCLUDE statement incorporates source code from an external library into the source program.

%INCLUDE searches all libraries in the LIBDEF SOURCE search chain specified in the JCL (for more information, see the *PL/I VSE Programming Guide).*

The VSE/Librarian format for a PL/I source member name is *member.type*, where *member* is 1-8 characters and *type* is a single character, defaulting to P.

Do not put other statements on the same line as the %INCLUDE statement.

The syntax for the %INCLUDE statement is:



type The member type within a library.

If you omit *type*, the default is P.

member

The name of the library member to be incorporated.

%INCLUDE statements can be nested; included source code can contain %INCLUDE statements.

For example, if the source program contains the following statement:

```
% INCLUDE PAYRL;
```

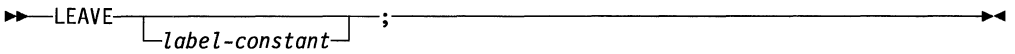
the following example is generated:

```
/* BEGIN %INCLUDE P(PAYRL)*******************************************/
DECLARE 1 PAYROLL,
          2 NAME,
            3 LAST CHARACTER(30) VARYING,
            3 FIRST CHARACTER(15) VARYING,
            3 MIDDLE CHARACTER(3) VARYING,
          2 HOURS,
            3 REGULAR FIXED DECIMAL(8,2),
            3 OVERTIME FIXED DECIMAL(8,2),
          2 RATE LIKE HOURS;
/* END    %INCLUDE P(PAYRL)*******************************************/
```

the structure declaration for PAYROLL is inserted into the source program. In this way, a central library of declarations can be used.

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. If the LEAVE statement is contained within a complex statement, control is transferred to the first statement following the termination of the complex statement. 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.

Examples of LEAVE statements

In the following example, the LEAVE statement transfers control to *next statement*:

```
DO  . . . ;
 .
 .
 .
LEAVE;
 .
 .
 .
END;
next statement;
```

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;
```

If the do-group does not have an explicit END statement, control is transferred as though all the END statements were present. For example:

```
A:  DO I = 1 TO 10;
    B:  DO J = 1 TO 5;
        IF X(I,J)=0 THEN
           LEAVE;
        ELSE  . . . ;
    END A;
```

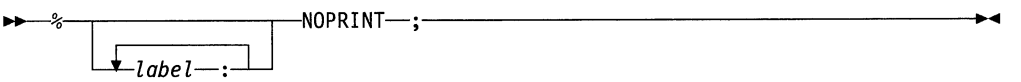
The LEAVE statement causes control to leave group B. The next iteration of group A, if there is one, then begins.

LOCATE statement

The LOCATE statement is described in "LOCATE statement" on page 254.

%NOPRINT statement

The %NOPRINT statement causes printing of the source and insource listings to be suspended until a %PRINT statement is encountered. The syntax for the %NOPRINT statement is:



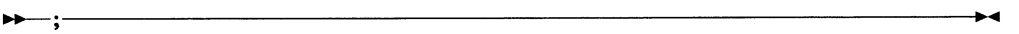
The %NOPRINT statement must be on a line with no other statements. It must not appear within another statement.

%NOTE statement

The %NOTE statement is described in "%NOTE statement" on page 427.

null statement

The null statement does nothing and does not modify sequential statement execution. It is often used to denote null action for THEN and ELSE clauses and WHEN and OTHERWISE statements. The syntax for the null statement is:



%null statement

The %null statement is described in "%null statement" on page 428.

ON statement

The ON statement is described in "ON statement" on page 320.

OPEN statement

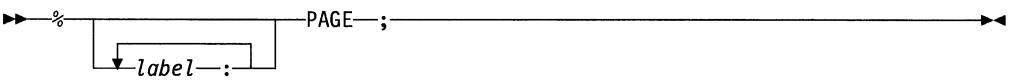
The OPEN statement is described in "OPEN statement" on page 244.

OTHERWISE statement

The OTHERWISE statement is described in this chapter under "SELECT statement" on page 194.

%PAGE statement

The statement following a %PAGE statement in the program listing is printed on the first line (after the page headings) of the next page. The syntax for the %PAGE statement is:



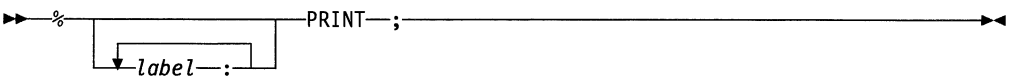
This statement controls both the insource and the source listing.

For paging to take place, the %PAGE statement must be on a line with no other statements.

When paging takes place, %PAGE does not appear in the formatted listing.

%PRINT statement

The %PRINT statement causes printing of the source and insource listings to be resumed. The syntax for the %PRINT statement is:



%PRINT is in effect at the onset of both the insource and the source listings, provided that the relevant compile-time options are specified.

The %PRINT statement must be on a line with no other statements. It must not appear within another statement.

PROCEDURE statement

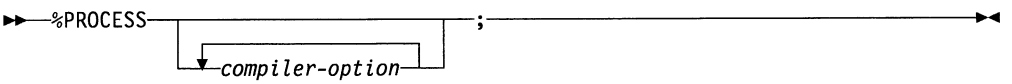
The PROCEDURE statement is described in "PROCEDURE statement" on page 110.

%PROCEDURE statement

The %PROCEDURE statement is described in "%PROCEDURE statement" on page 417.

%PROCESS statement

The %PROCESS statement is used to override compiler options.



The % must be the first data position of a source record. Any number of %PROCESS statements can be specified, but they must all appear before the first language element appears. Refer to the *PL/I VSE 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 "%PROCESS statement."

PUT statement

The PUT statement is described in "PUT statement" on page 269.

READ statement

The READ statement is described in "READ statement" on page 253.

RELEASE statement

The RELEASE statement is described in "RELEASE statement" on page 124.

RETURN statement

The RETURN statement for procedures is described in "RETURN statement" on page 148.

The preprocessor RETURN statement is described in "Preprocessor RETURN statement" on page 418.

REVERT statement

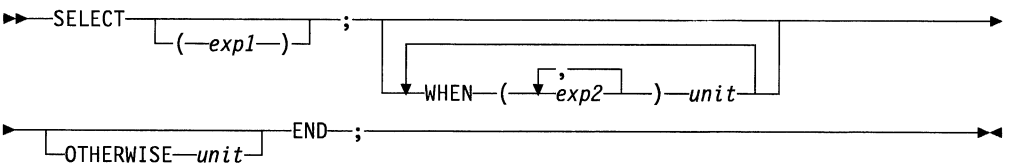
The REVERT statement is described in "REVERT statement" on page 324.

REWRITE statement

The REWRITE statement is described in "REWRITE statement" on page 254.

SELECT statement

A select-group provides a multi-way conditional branch. A select-group contains a SELECT statement, optionally one or more WHEN statements, optionally an OTHERWISE statement, and an END statement. The syntax for the select-group is:



Abbreviation: OTHER for OTHERWISE

SELECT
SELECT 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 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 prefix.

OTHERWISE
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 prefix.

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, ENTRY, 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 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 operands cannot be used in either *exp1* or *exp2*.

Examples of select-groups

In the following example, E, E1, etc., 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 WHEN statement is performed; no further WHEN statement expressions are evaluated. If none of the expressions in the WHEN statements is 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;
NL: next statement;
```

An example of *exp1* being omitted is:

```
SELECT;
  WHEN (A>B) CALL BIGGER;
  WHEN (A=B) CALL SAME;
  OTHERWISE CALL SMALLER;
END;
```

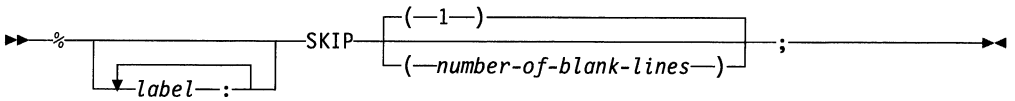
If a select-group contains no WHEN statements, the action in the OTHERWISE statement is executed unconditionally. If the OTHERWISE statement is omitted, and execution of the select-group does not result in the selection of a WHEN statement, the ERROR condition is raised.

SIGNAL statement

The SIGNAL statement is described in "SIGNAL statement" on page 325.

%SKIP statement

The specified number of lines following a %SKIP statement in the program listing are left blank. The syntax for the %SKIP statement is:



number-of-blank-lines
The number of lines to be skipped. Must be an integer in the range 1 through 999. Default is 1. If *number-of-blank-lines* 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.

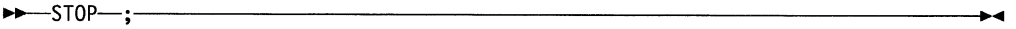
This statement controls both the insource and the source listing.

For skipping to take place, the %SKIP statement must be on a line with no other statements.

When skipping takes place, %SKIP does not appear in the formatted listing.

STOP statement

The STOP statement immediately terminates the program. The syntax for the STOP statement is:



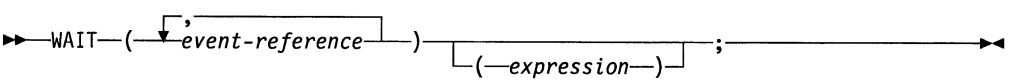
Prior to any termination activity, the FINISH condition is raised. On normal return from the FINISH ON-unit, the program terminates.

UNLOCK statement

The UNLOCK statement is described in "UNLOCK statement" on page 255.

WAIT statement

The execution of a WAIT statement within an activation of a block retains control for that activation of that block within the WAIT statement until specified events have completed. Control for a given block activation remains within the statement until, at possibly separate times during the execution of the statement, the completion value has been set complete for some or all of the event variables in the list. If an ON-unit entered due to the WAIT is terminated abnormally, control might not pass to the statement following the WAIT. The syntax for the WAIT statement is:



event-reference
An element or an array.

expression
Evaluated and converted to FIXED BINARY (31,0) when the WAIT statement is executed. This value specifies the number of events in the list that must be set complete before control for the block passes to the statement following the WAIT.

If the value is zero or negative, the WAIT statement is treated as a null statement. If the value is greater than the number, *n*, of event names in the list, the value is *n*. If the statement refers to an aggregate event variable, each of the elements contributes to the count.

If the expression does not appear, all the event variables in the list must be set complete before control is passed to the next statement in this block following the WAIT.

Event variables and the WAIT statement for record-oriented transmission are described under "EVENT option" on page 259.

If an abnormal return is made from any ON-unit entered from a WAIT, the associated event variable is set complete, the execution of the WAIT is terminated, and control passes to the point specified by the abnormal return.

WHEN statement

The WHEN statement is described in this chapter under "SELECT statement" on page 194.

WRITE statement

The WRITE statement is described in "WRITE statement" on page 254.

Chapter 8. Storage control

Chapter 8. Storage control

This chapter describes how you can control the allocation of storage.

All variables require storage. This includes problem data and program control data. The attributes specified for a variable describe the amount of storage required and how it is interpreted. For example:

```
DCL X FIXED BINARY(31,0) AUTOMATIC;
```

Storage allocation is the process of associating an area of storage with a variable so that the data item (or items) represented by the variable can be recorded internally. When storage is associated with a variable, the variable is *allocated*. A variable can be allocated either *statically* (before the execution of the program), or *dynamically* (during execution). A variable that is allocated statically remains allocated for the duration of the program. A variable that is allocated dynamically relinquishes its storage either upon the termination of the block containing that variable or at your request, depending upon its storage class.

The degree of storage control and the manner in which storage is allocated for a variable are determined by the storage class of that variable. There are four storage classes: static, automatic, controlled, and based. Each storage class is specified by its corresponding storage class attribute:

AUTOMATIC
Specifies that storage is allocated upon each entry to the block containing 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 121.)

STATIC Specifies that storage is allocated when the program is loaded. The storage is not freed until program execution is completed. For fetched procedures, the storage is not freed until the procedure is released.

CONTROLLED
Specifies that you maintain full control over the allocation and freeing of storage with the ALLOCATE and FREE statements. Multiple allocations of the same controlled variable, without intervening freeing, will stack generations of the variable.

BASED Specifies that you maintain full control over storage allocation and freeing (like CONTROLLED). Multiple allocations are not stacked but are available at any time. Each allocation can be identified by the value of a locator variable.

Storage class attributes can be declared explicitly for element, array, and major structure variables. For array and major structure variables, the storage class declared for the variable applies to all of the elements in the array or structure. Storage class attributes cannot be specified for entry constants, file constants, members of structures, or defined data items.

The default storage class is AUTOMATIC for internal variables and STATIC for external variables.

Automatic and based variables can have internal scope only. Static and controlled variables can have internal or external scope.

Parameters can be declared explicitly with the storage class attribute CONTROLLED. They cannot be declared explicitly with STATIC, BASED, or AUTOMATIC.

At no point in a PL/I program do you have access to the absolute address of a variable within main storage, because the allocation of storage for variables is managed by the compiler. You do not specify where in storage the allocation is to be made. You can, however, specify where it is to be allocated relative to storage already allocated—for instance, by allocating based variables in an area variable.

The chapter ends with descriptions of the DEFINED, POSITION, CONNECTED, and INITIAL attributes.

LE/VSE considerations

LE/VSE controls the allocation of storage for PL/I programs, and you can use LE/VSE run-time options to specify the amount of storage to be reserved for the different storage classes, and the initial value of the storage. PL/I storage classes AUTOMATIC, CONTROLLED, and BASED are allocated in LE/VSE storage as follows:

AUTOMATIC
LE/VSE maintains a *stack* for procedure-related storage items. PL/I automatic variables are allocated in the LE/VSE stack. The STACK run-time option controls the amount of stack storage to be reserved for automatic variables and other procedure-related items.

CONTROLLED
PL/I controlled variables are allocated in the LE/VSE *heap*. The amount of heap storage to be reserved can be controlled by the HEAP run-time option.

BASED If a based variable is allocated with an ALLOCATE statement that does not specify or imply an AREA variable, the based variable will be located in heap storage in the same manner as controlled variables.

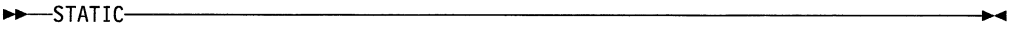
If the based variable is allocated in an AREA variable, it will inherit the storage class and location of the containing AREA variable.

If the based variable is never allocated specifically, it will inherit the storage class and location of the base data that its locator variable is pointing to.

For more information on the LE/VSE run-time options, see the *LE/VSE Programming Guide.*

Static storage and attribute

You use static storage when the variable is local to the procedure and the value it contains must be saved between successive invocations. Variables declared with the STATIC attribute are allocated prior to running a program. They remain allocated until the program terminates (STATIC storage in fetched procedures is an exception). The program has no control on the allocation of static variables during execution. The syntax for the STATIC attribute is:



Static variables follow normal scope rules for the validity of references to them. For example:

```
A: PROC OPTIONS(MAIN);
   .
   .
   .
   B: PROC;
      DECLARE X STATIC INTERNAL;
      .
      .
      .
      END B;
END A;
```

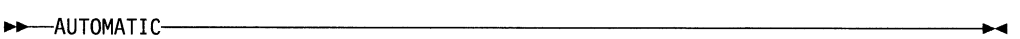
Although the variable X is allocated throughout the program, it can be referenced only within procedure B or any block contained in B.

If static variables are initialized using the INITIAL attribute, the initial values must be specified as constants (arithmetic constants can be optionally signed) with the exception of locator variables as noted below. Any specification of lengths, sizes, or bounds must be integers.

You can initialize a STATIC pointer or offset variable only by using the NULL or SYSNULL built-in function.

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

For example:

```
A:PROC;
  .
  .
  .
  CALL B;
  B:PROC;
    DECLARE (X,Y) AUTO;
    .
    .
    .
    END B;
  .
  .
  .
  CALL B;
```

Each time procedure B is invoked, the variables X and Y are allocated storage. When B terminates the storage is released, and the values they contain are lost. 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.

Array bounds, string lengths, and area sizes for automatic variables can be specified as expressions. This means that you can allocate a specific amount of storage when you need it. For example:

```
A:PROC;
  DECLARE N FIXED BIN;
  .
  .
  .
  B:PROC;
    DECLARE STR CHAR(N);
```

The character string STR has a length defined by the value of the variable N that existed when procedure B was invoked. If the declare statements are located in the same procedure, the compiler requires that the variable N be initialized either to a constant or to an initialized static variable. For example:

```
DCL N FIXED BIN (15) INIT(10),
    M FIXED BIN (15) INIT(N),
    STR1 CHAR(N),
    STR2 CHAR(M);
```

The length allocated is correct for STR1, but not for STR2. The compiler does not resolve this type of declaration dependency.

Controlled storage and attribute

Variables declared as CONTROLLED are allocated only when they are specified in an ALLOCATE statement. You have individual control over each controlled variable. A controlled variable remains allocated until a FREE statement that names the variable is encountered or until the end of the program in which it is allocated.

The scope of a controlled variable, when it is declared internal, is the block in which it is declared and any contained blocks. Any reference to a controlled variable that is not allocated produces undefined results.

The declaration of a controlled variable describes how much storage is required when the variable and its attributes are allocated. The syntax for the CONTROLLED attribute is:

Abbreviation: CTL

For example:

```
A:PROC;
  DCL X CONTROLLED;
  CALL B;
  .
  .
  .
    B:PROC;
    ALLOCATE X;
    .
    .
    .
    END B;
END A;
```

The variable X can be validly referred to within procedure B and that part of procedure A that follows execution of the CALL statement.

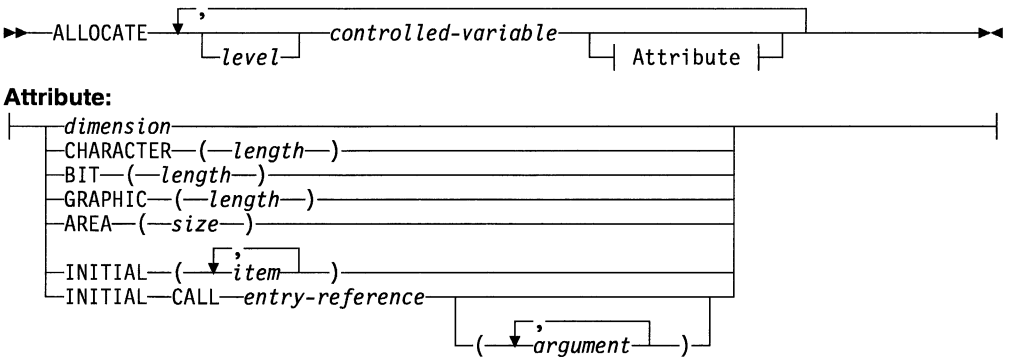
Generally, controlled variables are useful when a program requires large data aggregates with adjustable extents. For example:

```
DCL A(M,N) CTL;
GET LIST(M,N);
ALLOCATE A;
GET LIST(A);
  .
  .
  .
FREE A;
```

These statements allocate the exact storage required depending on the input data and free the storage when no longer required. 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. The bounds of controlled arrays, the lengths of controlled strings, and the size of controlled areas, as well as their initial values, can also be specified at the time the ALLOCATE statement is executed. The syntax for the ALLOCATE statement for controlled variables is:



Abbreviation: ALLOC

level A level number. The first name appearing after the keyword ALLOCATE must be a level-1 variable.

controlled-variable
A controlled variable or an element of a controlled major structure. A structure element, other than the major structure, can appear only if the relative structuring of the entire major structure containing the element appears as in the DECLARE statement for that structure. In this case, dimension attributes must be specified for all names that are declared with the dimension attribute.

Both controlled and based variables can be allocated in the same statement. See "ALLOCATE statement for based variables" on page 214 for the option syntax for based variables.

Bounds for arrays, lengths of strings, and sizes of areas are fixed at the execution of an ALLOCATE statement:

- If a bound, length, or size is explicitly specified in an ALLOCATE statement, it overrides the specification in the DECLARE statement.
- If a bound, length, or size is specified by an asterisk in an ALLOCATE statement, the bound, length, or size is taken from the current generation. If no generation of the variable exists, the bound, length, or size is undefined and the program is in error.
- The ALLOCATE, DECLARE, or DEFAULT statements must specify any necessary dimension, size, or length attributes for a variable. Any expression taken from a DECLARE or DEFAULT statement is evaluated at the point of allocation using the conditions defined at the ALLOCATE statement. However, names in the expression refer to variables whose scope includes the DECLARE or DEFAULT statement.
- In either an ALLOCATE or a DECLARE statement, if bounds, lengths, or sizes are specified by expressions that refer to the variable being allocated, the expressions are evaluated using the value of the most recent generation of the variable. For example:

```
DCL X(20) FIXED BIN CTL;
ALLOCATE X;
ALLOCATE X(X(1));
```

In the first allocation of X, the upper bound is specified by the DECLARE statement, that is, 20. In the second allocation, the upper bound is specified by the value of the first element of the first generation of X.

The dimension attribute must specify the same number of dimensions as declared. The dimension attribute can appear with any of the other attributes and must be the first attribute specified. For example:

```
DCL X(20) CHAR(5) CONTROLLED;
ALLOCATE X(25) CHAR(6);
```

The attributes BIT, CHARACTER, GRAPHIC, and AREA can appear only for variables having the same attributes, respectively.

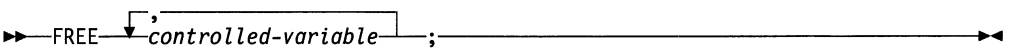
Initial values are assigned to a variable upon allocation, if it has an INITIAL attribute in either the ALLOCATE statement or DECLARE statement. Expressions or a CALL option in the INITIAL attribute are executed at the point of allocation, using the conditions enabled at the ALLOCATE statement, although the names are interpreted in the environment of the declaration. If an INITIAL attribute appears in both DECLARE and ALLOCATE statements, the INITIAL attribute in the ALLOCATE statement is used. If initialization involves reference to the variable being allocated, the reference is to the new generation of the variable.

A parameter that is declared CONTROLLED can be specified in an ALLOCATE statement.

Any evaluations performed at the time the ALLOCATE statement is executed (for example, evaluation of expressions in an INITIAL attribute) must not be interdependent.

FREE statement for controlled variables

The FREE statement frees the storage allocated for controlled variables. The storage can then be used for other allocations. For controlled variables, the next most recent allocation is made available, and subsequent references refer to that allocation. The syntax for the FREE statement for controlled variables is:



controlled-variable
A level-1, unsubscripted variable.

Both based and controlled variables can be freed in the same statement. See "FREE statement for based variables" on page 215 for the option syntax for based variables.

If a controlled variable has no allocated storage at the time the FREE statement is executed, no action is taken.

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, storage is *popped up* or removed from the stack. This is similar to the process described for automatic variables in a recursive procedure. For controlled variables, however, stacking and unstacking of variables occur at ALLOCATE and FREE statements rather than at block boundaries and are independent of invocation of procedures.

Asterisk notation

In an ALLOCATE statement, values are inherited from the most recent previous generation when dimensions, lengths, or sizes are indicated by asterisks. For arrays, the asterisk must be used for every dimension of the array, not just one of them. For example:

```
DCL X(10,20) CHAR(5) CTL;

ALLOCATE X;
ALLOCATE X(10,10);
ALLOCATE X(*,*);
```

The first generation of X has bounds (10,20); the second and third generations have bounds (10,10). The elements of each generation of X are all character strings of length 5.

The asterisk notation can also be used in a DECLARE statement, but has a different meaning. For example:

```
DCL Y CHAR(*) CTL,
      N FIXED BIN;

      N=20;
      ALLOCATE Y CHAR(N);
      ALLOCATE Y;
```

The length of the character string Y is taken from the previous generation unless it is specified in an ALLOCATE statement, in which case Y is given the specified length. This allows you to defer the specification of the string length until the actual allocation of storage.

Controlled structures

When a structure is controlled, any arrays, strings, or areas it contains can be adjustable. For this reason, you are allowed to describe the relative structuring in an ALLOCATE statement. For example:

```
DCL 1 A CTL,
      2 B(-10:10),
      2 C CHAR(*) VARYING;

ALLOCATE 1 A,
           2 B(1:10),
           2 C CHAR(5);
FREE A;
```

When the structure is allocated, A.B has the extent 1 to 10 and A.C is a VARYING character string with maximum length 5. When the structure is freed, only the major structure name is given. All of a controlled structure must be freed or allocated. An attempt to obtain storage for part of a structure is an error.

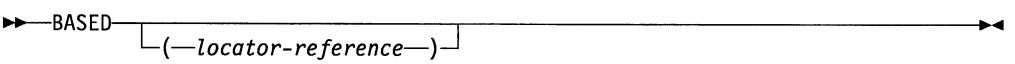
Built-In functions for controlled variables

When the allocation and freeing of a variable depend on flow of control, you can determine if a variable is allocated. The ALLOCATION built-in function returns a binary value of precision (31,0) indicating the number of generations that you can access for a given controlled variable. If the variable is not allocated, the function returns the value zero.

Other built-in functions that can be useful are the array-handling functions DIM, which determines the extent of a specified dimension of an array, and LBOUND and HBOUND, which determine the lower and upper bound, respectively, of a specified dimension of a given array. The CURRENTSTORAGE and STORAGE built-in functions return the amount of storage required by a particular variable. For strings, the built-in function LENGTH returns the current length of the string.

Based storage and attribute

A declaration of a based variable is a description of the generation; that is, 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 the value of a based variable that is not allocated produces undefined results. The syntax for the BASED attribute is:



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 based variable reference can be qualified by an external locator variable.

A based structure 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 215.

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 describe existing data, to obtain storage by means of the ALLOCATE statement, or to access data in a buffer by means of the LOCATE statement or READ (with SET option) statement.

You can use the ALLOCATE statement for a based variable. Because an independent 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. For example:

```
DCL X FIXED BIN BASED(P);
```

This declares that references to X, except when the reference is explicitly qualified, use the variable P to locate the storage for X.

The association of a locator-reference in this way is not a special relationship. 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, A(2,1), 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 (A(2,1) in the example).

This overlay technique can be achieved by use of the DEFINED attribute, but an important difference is that the overlay is permanent. When based variables are overlaid, the association can be changed at any time in the program by assigning a new value to the locator variable.

Although PL/I does not allow the overlay of variables with different attributes or precision (for example, overlaying a character string with a bit string), it is possible in this implementation. However, you should understand that this type of programming is an invalid use of PL/I and run-time results might be incorrect.

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 effectively 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 relative to the start of 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 any other data type, but pointer can be converted to offset, and vice versa.

When an offset variable is used in a reference, its value is implicitly converted to a pointer value. The address value of the area variable designated in the OFFSET attribute is added to the offset value. Explicit conversion of an offset to a pointer value is accomplished using the POINTER built-in function. For example:

```
DCL P POINTER, O OFFSET(A),B AREA;
P = POINTER(O,B);
```

This statement assigns a pointer value to P, giving the location of a based variable, identified by offset O in area 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 complements the POINTER built-in function and 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 effectively deducting the pointer value for the start of the area from the pointer value to be converted. This conversion is limited to pointer values that relate to addresses within the area named in the OFFSET attribute. Except when assigning the NULL built-in function value, it is an error to attempt to convert to an offset variable that is not associated with an area.

In conversion of offset data to pointer, the offset value is added to the pointer value of the start of the area named in the OFFSET attribute. It is an error to attempt to convert an offset variable that is not associated with an area.

In any conversion of locator data, if the offset variable is a member of a structure, or if it appears in a do-specification or a multiple assignment statement, the area associated with that offset variable must be an unsubscripted, nondefined, element variable.

The area can be based, but if so, its qualifier must be an unsubscripted, nonbased, nondefined pointer; this pointer must not be used to explicitly qualify the area in declaration of the offset variable.

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.

You can use a locator reference:

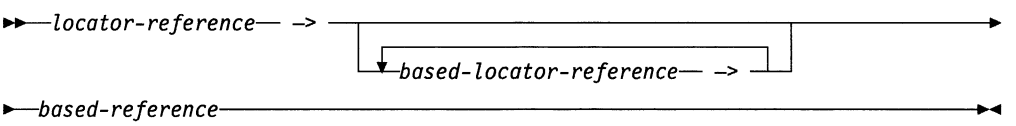
- As a locator qualifier, in association with a declaration of a based variable
- In a comparison operation, as in an IF statement (locator values can be compared as equal or not equal)
- As an argument in a procedure reference

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*.

Reference to a based variable can be explicitly qualified as shown in the following syntax:



For example:

```
P -> X
```

X is a based variable and P is a locator variable. The reference means "the generation of X identified by the value of the locator P." X is said to be *explicitly locator-qualified*.

When more than one locator qualifier is used in a reference, only the first, or leftmost, can be a function reference. All other locator qualifiers must be based references. However, an entry variable can be based, and can represent a function that returns a locator value.

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. For example:

```
DCL X FIXED BIN BASED(P);
ALLOCATE X;
X = X + 1;
```

The ALLOCATE statement sets a value in the pointer variable P so that the reference X applies to allocated storage. The references to X in the assignment statement are *implicitly locator-qualified* by P. References to X can also be explicitly locator-qualified as follows:

```
P->X = P->X + 1;
Q = P;
Q->X = Q->X + 1;
```

These assignment statements have the same effect as the previous example.

Because the locator declared with a based variable can also be based, a chain of locator qualifiers can be implied. For example:

```
DECLARE (P(10),Q) POINTER,
         R POINTER BASED (Q),
         V BASED (P(3)),
         W BASED (R),
         Y BASED;
ALLOCATE R,V,W;
```

Given this declaration and allocation, the following are valid references:

```
P(3) -> V

V

Q -> R -> W

R -> W

W
```

The first two references are equivalent, as are the last three. Any reference to Y must include a qualifying locator variable.

Levels of locator qualification

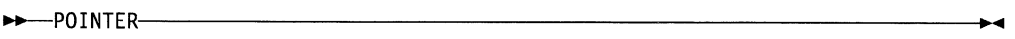
A pointer that qualifies a based variable represents one level of locator qualification. An offset represents two levels because it is implicitly qualified within an area. The number of levels is not affected by a locator being subscripted and/or an element of a structure. The maximum number of levels of locator qualification allowed in a reference depends on the available storage, but it will never be less than ten. For example:

```
DECLARE X BASED (P),
        P POINTER BASED (Q),
        Q OFFSET (A);
```

The references: X, P -> X, and Q -> P -> X all represent three levels of locator qualification.

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, or READ 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).

Setting pointer variables

Before a reference is made to a pointer-qualified variable, the pointer must have a value. A pointer value is obtained from any of the following:

- The NULL, SYSNULL, ADDR, ENTRYADDR, POINTERADD, POINTERVALUE, or POINTER built-in function
- A READ statement with the SET option
- An ALLOCATE or LOCATE statement
- By assignment of the value of another locator variable, or a locator value returned by a user-defined function

All pointer values are originally derived from one of these methods. These values can then be manipulated by the following:

- Assignment that copies a pointer value to a pointer variable
- Locator conversion that converts an offset value to a pointer value, or vice versa
- Passing the pointer value as an argument in a procedure reference
- Returning a pointer value from a function procedure

PL/I requires that all pointers declared in PL/I source be clean (that is, that the non-address bits in a fullword pointer are zero). Dirty pointers can result in unpredictable results, including program checks, incorrect pointer compare, mistaking a pointer for the NULL pointer, and incorrect last argument flagging.

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.

In general, the value of the NULL built-in function is used whenever a pointer (or offset) variable does not identify a location in storage. A pointer can be assigned the null value by:

- The NULL built-in function
- The ENTRYADDR built-in function of an unfetched procedure
- The value returned by the ADDR built-in function for an unallocated controlled variable

The SYSNULL built-in function can assign a pointer the system null value.

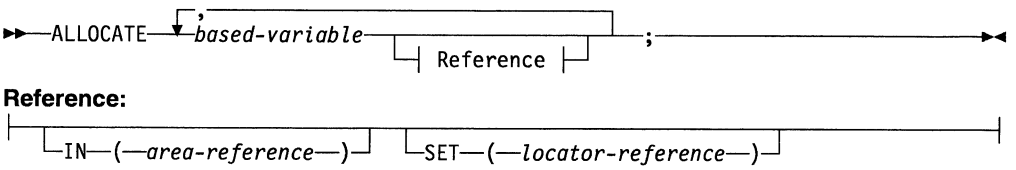
Note: NULL and SYSNULL do not compare equal, but you should not depend on them **not** being equal. In this PL/I release, NULL returns the value 'FF000000'X and SYSNULL returns the value '00000000'X.

Other useful built-in functions are the array-handling functions DIM, which determines the extent of a specified dimension of an array, and LBOUND and HBOUND, which determine the lower and upper bound of a specified dimension of a given array. The CURRENTSTORAGE and STORAGE built-in functions return

the amount of storage required by a particular variable. Similarly, for strings, the built-in function LENGTH returns the current length of the string.

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
An element variable, array, or major structure of any data type. If *based variable* is a major structure, you specify only the major structure name.

IN Specifies the area variable in which the storage is allocated.

SET Specifies a locator variable that is set to the location of the storage allocated. If the SET option is not specified, the declaration of the based variable must specify a locator variable.

You can allocate based and controlled variables in the same statement. See "ALLOCATE statement for controlled variables" on page 204 for the option syntax for controlled variables.

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 the AREA attribute.
- If the IN option is used *and* the declaration of the offset variable specifies the AREA attribute, the areas must be the same or one must contain the other.

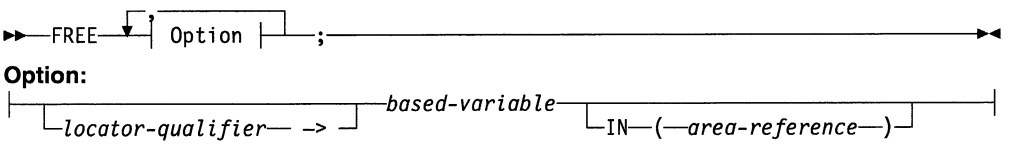
When an area is not used, the locator variable must be a pointer variable.

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 variable, and additional attributes cannot be specified in the ALLOCATE statement. Based variables are always allocated in multiples of 8 bytes.

A based structure can contain adjustable array bounds or string lengths or area sizes (see "REFER option (self-defining data)" on page 215). The asterisk notation for bounds, length, or size 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-qualifier ->
A particular generation of a based variable is freed by specifying a locator qualifier in the statement. If the based variable is not explicitly locator-qualified, the locator declared with the based variable 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. See "FREE statement for controlled variables" on page 206 for the option syntax for controlled variables.

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.

A FREE statement cannot be used to free a locate-mode I/O buffer.

Implicit freeing

A based variable need not be explicitly freed by a FREE statement, but it is 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 contains information about its own fields, such as the length of a string. A based structure can be declared to manipulate this data. String lengths, array bounds, and area sizes can all be defined by variables declared within the structure. When the structure is allocated (by either an ALLOCATE statement or a LOCATE statement), the value of an expression is

assigned to a variable that defines a length, bound, or size. For any other reference to the structure, the value of the defining variable is used.

The REFER option is used in the declaration of a based structure to specify that, on allocation of the structure, the value of an expression is assigned to a variable in the structure and represents the length, bound, or size of another variable in the structure. The syntax for a length, bound, or size with a REFER option is:

expression

When the based structure is allocated, the expression is evaluated and converted to FIXED BINARY (31,0). Any variables used as operands in the expression must not belong to the structure containing the REFER option.

variable

The variable, known as the *object* of the REFER option, must be a member of the declared structure. The REFER object must:

- Be REAL FIXED BINARY(p,0).
- Precede the first level-2 element that has the REFER option or contains an element that has the REFER option.
- Not be locator-qualified or subscripted.
- Not be part of an array.

For example:

```
DECLARE 1 STR BASED(P),
          2 X FIXED BINARY,
          2 Y (L REFER (X)),
        L FIXED BINARY INIT(1000);
```

This declaration specifies that the based structure STR consists of an array Y and an element X. 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 both the REFER option and the INITIAL attribute are used for the same member, initialization occurs after the object of the REFER has been assigned its value.

Any number of REFER options can be used in the declaration of a structure, provided that at least one of the following restrictions is satisfied:

- All objects of REFER options are declared at logical level two, that is, not declared within a minor structure. For example:

  ```
  DECLARE 1 STR BASED,
            2 (M,N),
            2 ARR(I REFER (M),
              J REFER(N)),
            2 X;
  ```

  When this structure is allocated, the values assigned to I and J will set the bounds of the two-dimensional array ARR.

- The structure is declared so that no padding between members of the structure can occur. "Structure mapping" on page 53 describes the rules for mapping structures. For example:

```
DECLARE 1 STR UNAL BASED (P),
          2 B FIXED BINARY,
          2 C,
            3 D FLOAT DECIMAL,
            3 E (I REFER (D))
              CHAR(J REFER (B)),
          2 G FIXED DECIMAL;
```

  All items require only byte alignment because this structure has the UNALIGNED attribute. Therefore, regardless of the values of B and D (the REFER objects) no padding occurs. Note that D is declared within a minor structure.

- If the REFER option is used only once in a structure declaration, the two preceding restrictions can be ignored provided that:
  - For a string length or area size, the option is applied to the last element of the structure.
  - For an array bound, the option is applied either to the last element of the structure or to a minor structure that contains the last element. The array bound must be the upper bound of the leading dimension. For example:

```
DCL 1 STR BASED (P),
      2 X FIXED BINARY,
      2 Y,
        3 Z FLOAT DECIMAL,
        3 M FIXED DECIMAL,
      2 D (L REFER (M)),
        3 E (50),
        3 F (20);
```

    The leading dimension of an array can be inherited from a higher level. If we had declared STR(4) in the above example, the leading dimension would be inherited from STR(4) and so it would not be possible to use the REFER option in D.

    This declaration does not satisfy the two previous bulleted restrictions. The REFER object M is declared within a minor structure and padding occurs. However, this restriction is satisfied because the REFER option is applied to a minor structure that contains the last element.

If the value of the object of a REFER option varies during the program then:

- The structure must not be freed until the object is restored to the value it had when allocated.
- The structure must not be written out while the object has a value greater than the value it was allocated.
- The structure can be written out when the object has a value equal to or less than the value it had when allocated. The number of elements, the string length, or area size actually written will be that indicated by the current value of the object. For example:

```
DCL 1 REC BASED (P),
      2 N,
      2 A (M REFER(N)),
    M INITIAL (100);
ALLOCATE REC;
N = 86;
WRITE FILE (X) FROM (REC);
```

86 elements of REC are written. It would be an error to attempt to free REC at this point, since N must be restored to the value it had when allocated (that is, 100). If N is assigned a value greater than 100, an error occurs when the WRITE statement is encountered.

When the value of a refer object is changed, the next reference to the structure causes remapping. For example:

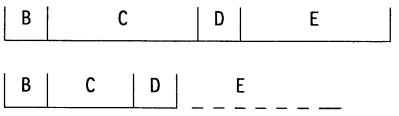
```
DCL 1 A BASED(P),
      2 B,
      2 C (I REFER(B)),
      2 D,
I INIT(10);

ALLOCATE A;
B = 5;
```

The next reference to A after the assignment to B remaps the structure to reduce the upper bound of C from 10 to 5, and to allocate to D storage immediately following the new last element of C. Although the structure is remapped, no data is reassigned—the contents of the part of storage originally occupied by the structure A are unchanged. If you do not consider remapping, errors can occur.

Consider the following example, in which there are two REFER options in the one structure:

```
DCL 1 A BASED (P),
      2 B FIXED BINARY (15,0),
      2 C CHAR (I1 REFER (B)),
      2 D FIXED BINARY (15,0),
      2 E CHAR (I2 REFER (D)),
(I1,I2) INIT (10);

ALLOCATE A;
B = 5;
```

The mapping of A with B=10 and B=5, respectively, is as follows:



D now refers to data that was originally part of that assigned to the character-string variable C. This data is interpreted according to the attributes of D—that is, as a fixed-point binary number—and the value obtained will be the length of E. Hence, the length of E is unpredictable.

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.

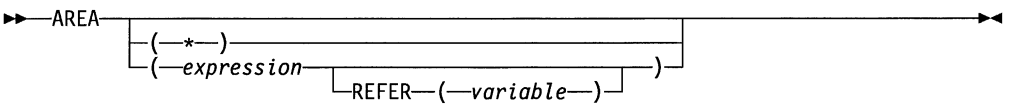
When a based variable is allocated and an area is not specified, the storage is obtained from wherever it is available. Consequently, allocated based variables can be scattered widely throughout main storage. This is not significant for internal operations because items are readily accessed using the 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 or 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 *, is not specified, the default is 1000.

***** An asterisk can be used to specify the size if the area variable declared is controlled or is a parameter. If a controlled area variable is declared with an asterisk, the size must be specified in the ALLOCATE statement used to allocate the area. If a parameter is declared with an asterisk, the size is inherited from the argument.

REFER See "REFER option (self-defining data)" on page 215 for the description of the REFER option.

The area size for areas that have the storage classes AUTOMATIC or CONTROLLED is given by an expression whose integer value specifies the number of reserved bytes.

If an area has the BASED attribute, the area size must be an integer unless the area is a member of a based structure and the REFER option is used.

The size for areas of static storage class must be specified as an integer.

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.

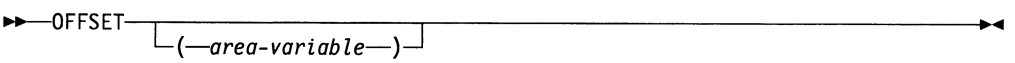
When a based variable is freed, the storage it occupied is available for other allocations. A chain of available storage within an area is maintained; the head of the chain is held within the 16 bytes of control information. Inevitably, as based variables with different storage requirements are allocated 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 a special relationship. An offset variable can be associated with any area variable by means of the POINTER built-in function (see "Locator conversion" on page 210). 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
- By assignment of the NULL built-in function value

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;

NEXT: ALLOCATE STR SET(T->P);
    T=T->P;
    .
    .
    GO TO NEXT;
```

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. For 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

The area facility allows 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 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, V-format or U-format 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 and structures in PL/I 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 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 that includes a pointer variable. For example:

```
DCL 1 STR BASED(H),
      2 P POINTER,
      2 DATA,
    T  POINTER;

      ALLOCATE STR;
      T=H;

NEXT: ALLOCATE STR SET(T->P);
        T=T->P;
        T->P=NULL;
        .
        .
        .
        GO TO NEXT;
```

Here a list of structures is created. 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 10 shows a diagrammatic representation of a one-directional chain.

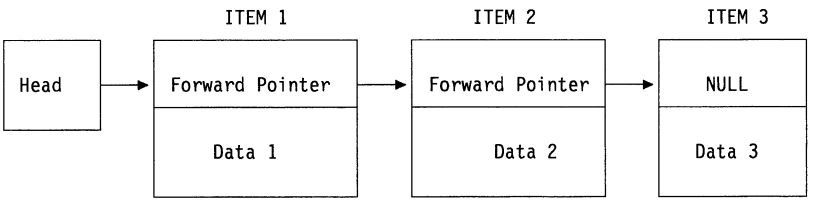


Figure 10. Example of one-directional chain

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 previous 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. 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.

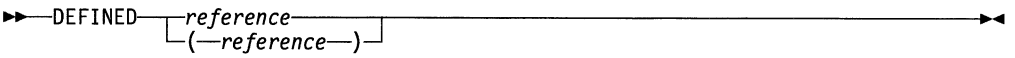
A list need not consist only of generations of a single based variable. Generations of different based structures can be included in a list by setting the appropriate pointer values. Items can be added and deleted from a list by manipulating the values of pointers. A list can be restructured by manipulating the pointers so that the processing of data in the list can be simplified.

DEFINED attribute

Table 21. Guide to types of defining

POSITION attribute specified	References to iSUB variables in base item subscripts	Base and defined match	Type of defining in effect
YES	NO	-	string overlay
NO	YES	-	iSUB
	NO	YES	simple
		NO	string overlay

The DEFINED attribute specifies that the declared variable is associated with some or all of the storage associated with the designated base variable. The syntax for the DEFINED attribute is:



Abbreviation: DEF

reference
to the variable (the *base variable*) whose storage is associated with the declared variable; the latter is the *defined variable*. The base variable can be EXTERNAL or INTERNAL. It can be a parameter (in string overlay defining, the parameter must refer to connected storage). It cannot be BASED or DEFINED. A change to the base variable's value is a corresponding change to the value of the defined variable, and vice versa.

If the base variable is a data aggregate, a defined variable can comprise all the data or only a specified part of it.

The defined variable does not inherit any attributes from the base variable. The defined variable must be INTERNAL and a level-1 identifier. It can have the dimension attribute. It cannot be INITIAL, AUTOMATIC, BASED, CONTROLLED, STATIC, or a parameter.

There are three types of defining: simple, iSUB, and string overlay.

If the POSITION attribute is specified, string overlay defining is in effect. In this case the base variable must not contain iSUB references. If the subscripts specified in the base variable contain references to iSUB variables, iSUB defining is in effect. If neither iSUB variables nor the POSITION attribute is present, simple defining is in effect if the base variable and defined variable match according to the criteria given below. Otherwise string overlay defining is in effect. For a tabulated summary of these rules, see Table 21.

A base variable and a defined variable *match* if the base variable when passed as an argument matches a parameter which has the attributes of the defined variable (except for the DEFINED attribute). For this purpose, the parameter is assumed to have all array bounds, string lengths, and area sizes specified by asterisks.

For simple and iSUB defining, a PICTURE attribute can only be matched by a PICTURE attribute that is identical except for repetition factors. For a reference to specify a valid base variable in string overlay defining, the reference must be in connected storage. You can override the matching rule completely, but this can cause unwanted side effects within your program.

The values specified or derived for any array bounds, string lengths, or area sizes in a defined variable do not always have to match those of the base variable. However, the defined variable must be able to fit into the corresponding base array, string, or area.

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.

The determination of values and the interpretation of names occurs in the following sequence:

1. The array bounds, string lengths, and area sizes of a defined variable are evaluated on entry to the block that declares the variable.
2. 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 creation of a dummy, the corresponding parameter refers to the generation of the base variable that is current when the argument is passed. This remains true even if the base variable is reallocated within the invoked procedure.
3. When a reference is made to the defined variable, the order of evaluation of the subscripts of the base and defined variable is undefined.

If the defined variable has the BIT attribute, unpredictable results can occur under the following conditions:

- If the base variable is not on a byte boundary
- If the defined variable is not defined on the first position of the base variable and the defined variable is used as:
  - A parameter in a subroutine call (that is, referenced as internally stored data)
  - An argument in a PUT statement
  - An argument in a built-in function (library call)
  - If the base variable is controlled, and the defined variable is dimensioned and is declared with variable array bounds

Unconnected storage

The DEFINED attribute can overlay arrays. This allows array expressions to refer to array elements in unconnected storage (array elements that are not adjacent in storage). It is possible for an array expression involving consecutive elements to refer to unconnected storage in the two following cases:

- Where an array is declared with iSUB defining. An array expression that refers to adjacent elements in an array declared with iSUB defining can be a reference to unconnected storage (that is, a reference to elements of an overlaid array that are not adjacent in storage).
- Where a string array is defined on a string array that has elements of greater length. Consecutive elements in the defined array are separated by the difference between the lengths of the elements of the base and defined arrays, and are held in unconnected storage.

An array overlay defined on another array is always assumed to be in unconnected storage.

See "CONNECTED attribute" on page 230 for information about the connected attribute.

Simple defining

Simple defining allows you to refer to an element, array, or structure variable by another name.

The defined and base variables can comprise any data type, but they must match, as described earlier. The ALIGNED and UNALIGNED attributes must match for each element in the defined variable and the corresponding element in the base variable.

The defined variable can have the dimension attribute.

In simple defining of an array:

- The base variable can be a cross-section of an array.
- The number of dimensions specified for the defined variable must be equal to the number of dimensions specified for the base variable.
- The range specified by a bound pair of the defined array must equal or be contained within the range specified by the corresponding bound pair of the base array.

In simple defining of a string, the length of the defined string must be less than or equal to the length of the base string.

In simple defining of an area, the size of the defined area must be equal to the size of the base area.

A base variable can be, or can contain, a varying string, provided that the corresponding part of the defined variable is a varying string of the same maximum length.

Examples:

```
DCL A(10,10,10),
    X1(2,2,2) DEF A,
    X2(10,10) DEF A(*,*,5),
    X3 DEF A(L,M,N);
```

X1 is a three-dimensional array that consists of the first two elements of each row, column and plane of A. X2 is a two-dimensional array that consists of the fifth plane of A. X3 is an element that consists of the element identified by the subscript expressions L, M, and N.

```
DCL B CHAR(10),
    Y CHAR(5) DEF B;
```

Y is a character string that consists of the first 5 characters of B.

```
DCL C AREA(500),
    Z AREA(500) DEF C;
```

Z is an area defined on C.

```
DCL 1 D UNALIGNED,
        2 E,
        2 F,
            3 G CHAR(10) VAR,
            3 H,
    1 S UNALIGNED DEF D,
        2 T,
        2 U,
            3 V CHAR(10) VAR,
            3 W;
```

S is a structure defined on D. For simple defining, the organization of the two structures must be identical. A reference to T is a reference to E, V to G, and so on.

iSUB defining

By defining an iSUB, you can create a defined array that consists of designated elements from a base array. The defined and base arrays can be arrays of structures, can comprise any data types, and must have identical attributes (not including the dimension attribute).

The defined variable must have the dimension attribute. In the declaration of the defined array, the base array must be subscripted. The subscript positions cannot be specified as asterisks.

An iSUB variable is a reference, in the subscript list for the base array, to the ith dimension of the defined array. At least one subscript in the base-array subscript-list must be an iSUB expression which, on evaluation, gives the required subscript in the base array. The value of i ranges from 1 to n, where n is the number of dimensions in the defined array. The number of subscripts for the base array must be equal to the number of dimensions for the base array.

If a reference to a defined array does not specify a subscript expression, subscript evaluation occurs during the evaluation of the expression or assignment in which the reference occurs.

The value of i is specified as an integer. Within an iSUB expression, an iSUB variable is treated as a REAL FIXED BINARY variable, with precision (31,0).

A subscript in a reference to a defined variable is evaluated even if there is no corresponding iSUB in the base-variable subscript list.

An iSUB-defined variable cannot appear in the explicit data-list of a data-directed transmission statement.

Examples:

```
DCL A(10,10),
    X(10) DEFINED (A(1SUB,1SUB));
```

X is a one-dimensional array that consists of a diagonal of A.

```
DCL B(2,5),
    Y(5,2) DEF B(2SUB,1SUB);
```

Y is a two-dimensional array that consists of the elements of B with the bounds transposed.

```
DCL A(10,10) ,  B(5,5) DEF
    A(1+1SUB/5,1+2SUB/5);
```

In this case there is a many-to-one mapping of certain elements of B to a single element of A. B(I,J) is defined on:

```
A(1,1) for I<5 and J<5
A(1,2) for I<5 and J=5
A(2,1) for I=5 and J<5
A(2,2) for I=5 and J=5
```

Since all the elements B(I,J) are defined on the single element A(1,1) when I<5 and J<5, assignment of a value to one of these elements causes the same value to be assigned to all of them.

String overlay defining

String overlay defining allows you to associate a defined variable with the storage for a base variable. Both the defined and the base variable must be string or picture data.

Neither the defined nor the base variable can have the ALIGNED or the VARYING attributes.

Both the defined and the base variables must belong to:

- The bit class, consisting of:

  Fixed-length bit variables
  Aggregates of fixed-length bit variables

- The character class, consisting of:

  Fixed-length character variables
  Character pictured and numeric pictured variables
  Aggregates of the two above

- The graphic class, consisting of:

  Fixed-length graphic variables
  Aggregates of fixed-length graphic variables

Examples:

```
DCL A CHAR(100),
    V(10,10) CHAR(1) DEF A;
```

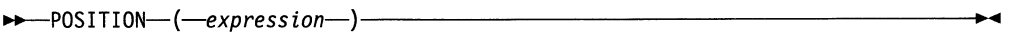
V is a two-dimensional array that consists of all the elements in the character string A.

```
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.

POSITION attribute

The POSITION attribute can be used only with string-overlay defining and specifies the bit, character, or graphic within the base variable at which the defined variable is to begin. The syntax for the POSITION attribute is:



Abbreviation: POS

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*, where *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 class aggregate:

- The POSITION attribute can contain only an integer.
- The base variable must not be subscripted.

The base variable must refer to data in connected storage.

Examples:

```
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 20th 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 decimal 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. Otherwise, simple defining is used and gives different results.

CONNECTED attribute

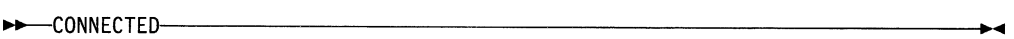
Elements, arrays, and major structures are always allocated in connected storage. References to unconnected storage arise only when you refer to an aggregate that is made up of noncontiguous items from a larger aggregate. (See "Cross sections of arrays" on page 47, and "DEFINED attribute" on page 224.) For example, in the structure:

```
1 A(10),
  2 B,
  2 C;
```

the interleaved arrays A.B and A.C are both in unconnected storage.

Certain restrictions apply to the use of unconnected storage. For example, a record variable (that is, a variable to or from which data is transmitted by a record-oriented transmission statement) must represent data in *connected storage*.

The CONNECTED attribute is applied only to parameters, and specifies that the parameter is a reference to connected storage only and, hence, allows the parameter to be used as a target or source in record-oriented I/O or as a base in string overlay defining. The syntax for the CONNECTED attribute is:



Abbreviation: CONN

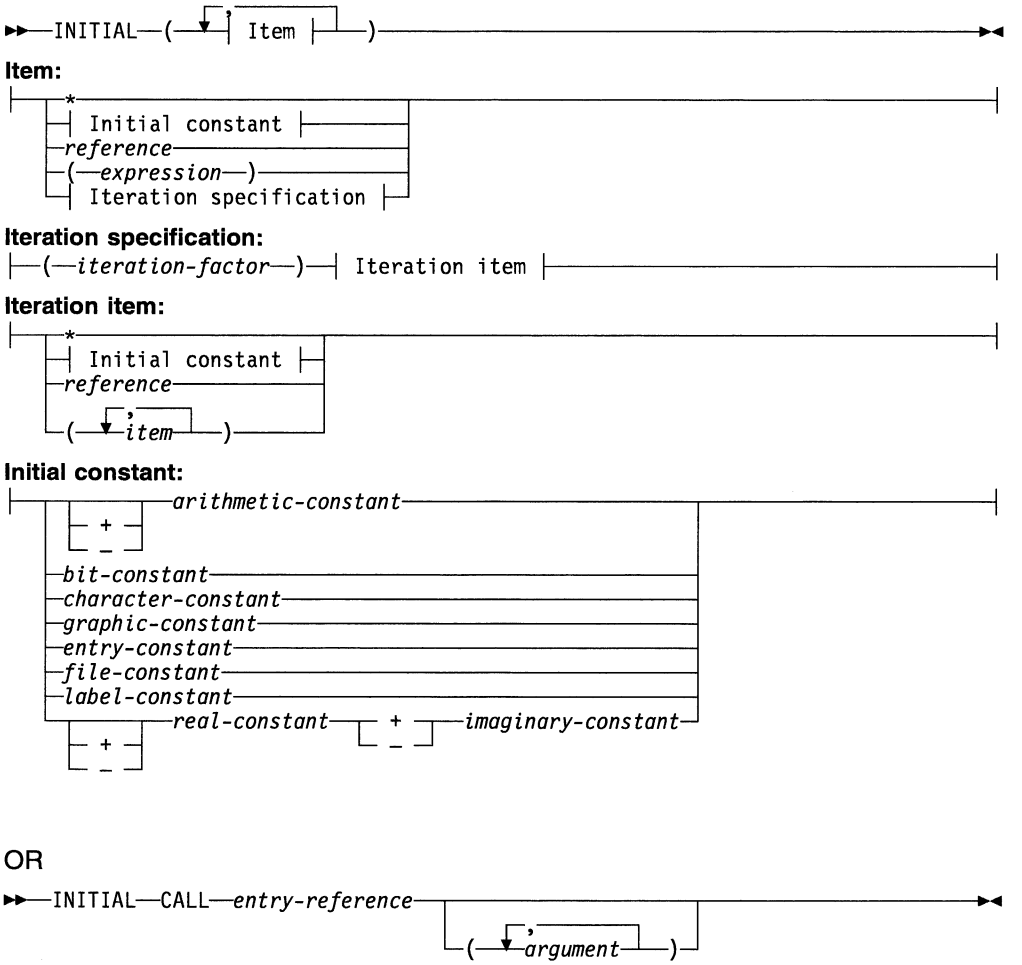
CONNECTED can be specified for noncontrolled aggregate parameters and can be associated only with level-one names.

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 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 for constants, defined data, parameters (except controlled parameters), or static entry variables.

The INITIAL attribute has two forms. The first specifies an initial constant, expression, or function reference, whose value is assigned to a variable when storage is allocated to it. The second form specifies that, through the CALL option, a procedure is invoked to perform initialization at allocation. The variable is initialized by assignment during the execution of the called routine (rather than by this routine being invoked as a function that returns a value to the point of invocation). The syntax for the INITIAL attribute is:



OR

Abbreviation: INIT

* Specifies that the element is to be left uninitialized.

iteration-factor

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, except for static data, in which case it must be an integer. When storage is allocated for the array, the expression is evaluated to give an integer that specifies the number of iterations. A negative or zero iteration-factor specifies no initialization.

constant
reference
expression
An initial value to be assigned to the initialized variable.

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.)

For automatic variables, which are allocated at each activation of the declaring block, any specified initial value is assigned with each allocation.

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. However, this initialization of controlled variables can be overridden in the ALLOCATE statement.

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.

Only constant values with no operations, for example, 3 or 'ABC', can be specified in the INITIAL attribute for static variables, except that the NULL built-in function can be used to initialize a static pointer variable.

Expressions enclosed in parentheses containing concatenated string constants can be used to initialize static string variables. For example:

```
DCL C CHAR(3) STATIC INIT (('A'||'BC'))
```

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.

For example:

```
((2)'A')     is equivalent to  ('AA')
((2)('A'))   is equivalent to  ('A','A')
((2)(1)'A') is equivalent to  ('A','A')
```

On allocation, an area variable is initialized with the value of the EMPTY built-in function, after which any specified INITIAL value is applied. You can initialize an area by assigning it the value of another area, using the INITIAL attribute with or without the CALL option.

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.

If a STATIC EXTERNAL item is given the INITIAL attribute in more than one declaration, the value specified must be the same in every case.

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 106.

INITIAL CALL cannot be used to initialize static data.

If the procedure invoked by the INITIAL CALL attribute has been specified in a FETCH or RELEASE statement and it is not present in main storage, the INITIAL CALL attribute initiates dynamic loading of the procedure from auxiliary storage (see "Dynamic loading of an external procedure" on page 122).

Static label variables cannot have the INITIAL attribute except when using the compiler to compile procedures containing STATIC LABEL arrays. In this case, improved performance can be obtained by specifying the INITIAL attribute.

If both the REFER option and the INITIAL attribute are used for the same member, initialization is done after the object of the REFER has been assigned its value.

If the variable has the REFER option and the item involves a base element or a substructure of the current generation of the variable, the result of the INITIAL attribute is undefined. For example:

```
DCL 1 A,
      2 B,
      2 C CHAR(N REFER(B))
              INIT('AAB'),
      2 D CHAR(5) INIT(C);
ALLOCATE A;
```

the result of initializing D is undefined.

For an alternate method of initializing arrays of nonstatic label variables, see "Label data and attribute" on page 39.

Examples of the INITIAL attribute

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 illustrates the use of a function reference to initialize a variable:

```
DECLARE X INIT(SQRT(Z));
```

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, SET_UP is the name of a procedure that can set the initial values of elements in TABLE. X and Y are arguments passed to SET_UP.

```
DECLARE TABLE (20,20) INITIAL
        CALL SET_UP (X,Y);
```

In the following example, only the first, third, and fourth elements of A are initialized; the rest of the array is uninitialized. The array B is fully initialized, with the first 25 elements initialized to 0, the next 25 to 1, and the last 50 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,(50)0),

        1 C(8),
          2 D INITIAL (0),
          2 E INITIAL((8)0);
```

When an array of structures is declared with the LIKE attribute to obtain the same structuring as a structure whose elements have been initialized, only the first structure is initialized. For example:

```
DECLARE 1 G,
          2 H INITIAL(0),
          2 I INITIAL(0),
        1 J(8) LIKE G;
```

Only J(1).H and J(1).I are initialized in the array of structures.

Chapter 9. Input and output

Chapter 9. Input and output

Both this chapter and the *PL/I VSE Programming Guide* discuss those aspects of PL/I input and output that are common to stream-oriented and record-oriented data transmission, including files and their attributes, and the relationship of files to data sets.

PL/I input and output statements let you transmit data between the main storage 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 *input*. Transmission of data from a program to a data set is *output*.

PL/I input and output statements are concerned with the logical organization of a data set, not 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 uses models of data sets, called *files*. A file can be associated with different data sets at different times during the execution of a program.

A PL/I program can use two types of data transmission: stream and record. The same data set can be processed at different times by either stream or record data transmission (but only if all items in the data set are in character form).

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. DBCS data is processed unchanged.

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 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. (This is not strictly true for ASCII data sets—see "Information interchange codes" on page 237.)

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.

The next two chapters describe the data transmission statements you can use in a PL/I program, and the *PL/I VSE Programming Guide* describes the various data set organizations that are recognized in PL/I.

Data sets

Data sets are stored on a variety of auxiliary storage media, such as magnetic tape and direct access storage devices. Despite their variety, these media have characteristics that allow common methods of collecting, storing, and transmitting data.

Data set organization

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.

These storage and retrieval methods provide PL/I with three data set organizations:

- CONSECUTIVE
- REGIONAL
- VSAM

The characteristics of the data set organizations available are described in the *PL/I VSE Programming Guide.*

If the data set organization is not specified in the ENVIRONMENT option, the default is CONSECUTIVE.

Information interchange codes

The code used to represent data, both in main storage and on auxiliary storage, is EBCDIC (extended binary-coded-decimal interchange code). In general, PL/I compiled programs use EBCDIC to record all character data.

The operating system also supports ASCII (American Standard Code for Information Interchange) to represent data on magnetic tape. PL/I can read and create ASCII data sets. The operating system translates between the two codes.

Apart from the options specified in the ENVIRONMENT attribute, the same PL/I program can handle either ASCII or EBCDIC data sets. On output, the operating system translates from EBCDIC to ASCII immediately before data is written from a buffer to external storage. On input, the operating system translates from ASCII to EBCDIC as soon as a buffer is full.

In PL/I, only CHARACTER data can be written onto an ASCII data set.

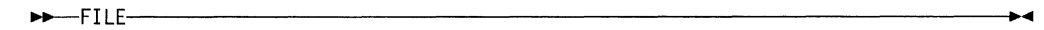
Use the ASCII and BUFOFF options of the ENVIRONMENT attribute if you are reading or writing data sets recorded in ASCII. (These are described in the *PL/I VSE Programming Guide.)*

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 uses 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 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*.

The attributes associated with a file constant fall into two categories:

alternative attribute An attribute 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.

additive attribute An attribute that must be stated explicitly or is implied by another explicitly stated attribute. The additive attribute KEYED is implied by the DIRECT attribute. The additive attribute PRINT can be implied by the output file name SYSPRINT.

The following lists show the attributes that apply to each type of data transmission:

Stream-oriented data transmission

ENVIRONMENT
INPUT
OUTPUT
PRINT
STREAM

Record-oriented data transmission

BACKWARDS	EXCLUSIVE[1]	RECORD
BUFFERED	INPUT	SEQUENTIAL
DIRECT	KEYED	UNBUFFERED
ENVIRONMENT	OUTPUT	UPDATE

Note:

1. Syntax checked only; has no effect. Kept for compatibility with PL/I MVS & VM.

Table 22 on page 239 shows the valid combinations of file attributes.

Table 22. Attributes of PL/I file declarations

Data set type	Stream	Record							Legend:
		Sequential					Direct		C Checked for VSAM
		Consecutive		Regional					D Default
File Type	Consecutive	Buffered	Unbuffered	Buffered	Unbuffered	VSAM	Regional	VSAM	I Must be specified or implied N Ignored for VSAM O Optional S Must be specified - Invalid
File attributes[1]									**Attributes implied**
File	I	I	I	I	I	I	I	I	
Input[1]	D	D	D	D	D	D	D	D	File
Output	O	O	O	O	O	O	O	O	File
Environment	I	I	I	S	S	S	S	S	File
Stream	D	-	-	-	-	-	-	-	File
Print[1]	O	-	-	-	-	-	-	-	File stream output
Record	-	I	I	I	I	I	I	I	File
Update[2]	-	O	O	O	O	O	O	O	File record
Sequential	-	D	D	D	D	D	-	D	File record
Buffered	-	D	-	D	-	D	-	S	File record
Unbuffered	-	-	S	-	S	S	D	D	File record
Backwards[3]	-	O	O	-	-	-	-	-	File record sequential input
Keyed[4]	-	-	-	O	O	O	I	O	File record
Direct	-	-	-	-	-	S	S	S	File record keyed
Exclusive[5]	-	-	-	-	-	-	-	-	

Notes:

1. A file with the INPUT attribute cannot have the PRINT attribute.
2. UPDATE is invalid for tape files.
3. BACKWARDS is valid only for input tape files.
4. Keyed is required for REGIONAL output.
5. Syntax checked only; has no effect. Kept for compatibility with PL/I MVS & VM.

In the following example, the name MASTER is declared a file constant:

```
DECLARE MASTER FILE;
```

File variable

A file variable has the attributes FILE and VARIABLE; it cannot have any of the file 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 for a name with the FILE attribute if the name is an element of an array or structure, or if any of the following attributes is specified:

Storage class attribute
parameter
alignment attribute
DEFINED
INITIAL

In the following statement, the name ACCOUNT is declared a file variable, and ACCT1, ACCT2, ... are declared file constants; the file constants can subsequently be assigned to the file variable.

```
DECLARE ACCOUNT FILE VARIABLE,
        ACCT1 FILE,
        ACCT2 FILE;
```

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 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 323). For example:

```
DCL F FILE,
    G FILE VARIABLE;
    G=F;
L1:  ON ENDFILE(G);
L2:  ON ENDFILE(F);
```

The statements labeled L1 and L2 both specify null ON-units for the same file.

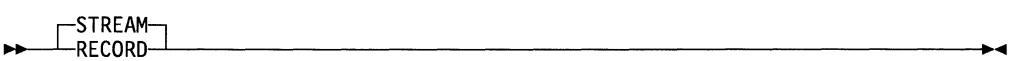
Alternative attributes

PL/I provides five groups of alternative file attributes. Each group (except scope, discussed in "Scopes of declarations" on page 154) is discussed individually below. The groups are:

Group type	Alternative attributes	Default attribute
Usage	STREAM|RECORD	STREAM
Function	INPUT|OUTPUT|UPDATE	INPUT
Access	SEQUENTIAL|DIRECT	SEQUENTIAL
Buffering	BUFFERED|UNBUFFERED	BUFFERED (for SEQUENTIAL files); UNBUFFERED (for DIRECT files)
Scope	EXTERNAL|INTERNAL	EXTERNAL

RECORD and STREAM attributes

The RECORD and STREAM attributes specify the kind of data transmission used for the file. The syntax for the RECORD and STREAM attributes is:



STREAM

Specifies 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.

A file with the STREAM attribute can be specified only in the OPEN, CLOSE, GET, and PUT input/output statements.

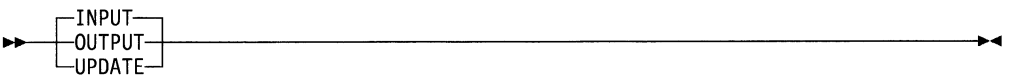
RECORD
Specifies that the file consists of a collection of physically separate records, each of which consists of one or more data items in any form. Each record is transmitted as an entity to or from a variable.

A file with the RECORD attribute can be specified only in the OPEN, CLOSE, READ, WRITE, REWRITE, LOCATE, and DELETE input/output statements.

The default is STREAM.

INPUT, OUTPUT, and UPDATE attributes

The function attributes specify the direction of data transmission allowed for a file. The syntax for the INPUT, OUTPUT, and UPDATE attributes is:



INPUT
Data is transmitted from auxiliary storage to the program.

OUTPUT
Data is transmitted from the program to auxiliary storage, to create a new data set.

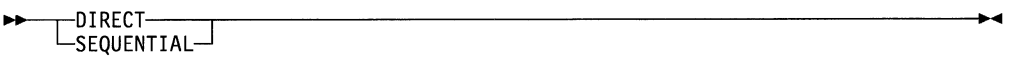
UPDATE
(RECORD files only.) Data can be transmitted in either direction: from the program to auxiliary storage, and from auxiliary storage to the program. UPDATE lets you insert new records in an existing data set and alter records already in the data set.

Specifying UPDATE for a SEQUENTIAL file indicates the update-in-place mode.

The default is INPUT.

SEQUENTIAL and DIRECT attributes

The 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

SEQUENTIAL
Specifies that:

- Records in a CONSECUTIVE or REGIONAL data set are accessed in physical sequence
- Records in a VSAM KSDS are accessed in key sequence order

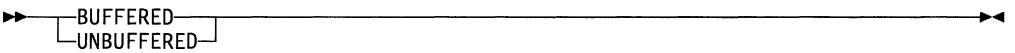
For certain data set organizations, a file with the SEQUENTIAL attribute can also be used for random access or for a mixture of random and sequential access. In this case, the file must have the additive attribute KEYED. Existing

records of a data set in a SEQUENTIAL UPDATE file can be modified, ignored, or, if the data set is a VSAM KSDS, deleted.

DIRECT
Specifies that records in a data set can be accessed in any order. The location of the record in the data set is determined by a character-string *key*; the DIRECT attribute implies the KEYED attribute. The associated data set must be on a direct-access volume.

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
UNBUF for UNBUFFERED

BUFFERED
Specifies that during transmission to and from auxiliary storage each record of a RECORD file must pass through intermediate storage buffers. If BUFFERED is specified, data transmission (in most cases) is overlapped with processing.

UNBUFFERED
Specifies 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. It does not, however, specify that records cannot be buffered. A buffer will, in fact, be used if REGIONAL(2) or REGIONAL(3) is specified in the ENVIRONMENT attribute or if the records are variable-length. A file with the UNBUFFERED attribute must not be blocked. When UNBUFFERED is specified, data transmission is not overlapped with processing. You must use the EVENT option to achieve such overlapping.

The default for SEQUENTIAL files is BUFFERED. UNBUFFERED is the default for DIRECT files unless BUFFERED is specified explicitly.

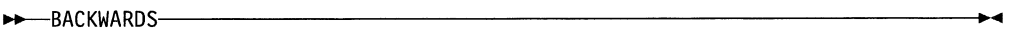
Additive attributes

The additive attributes are:

```
BACKWARDS
ENVIRONMENT
EXCLUSIVE
KEYED
PRINT
```

BACKWARDS attribute

The BACKWARDS attribute specifies that the records of a SEQUENTIAL RECORD INPUT file associated with a data set on magnetic tape are to be accessed in reverse order; that is, from the last record to the first record. A file cannot be opened backwards if the last operation performed on the tape was a forward read where the file was not read to end-of-file. The syntax for the BACKWARDS attribute is:

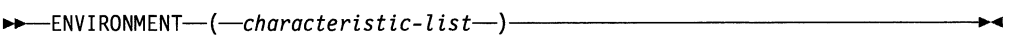


For unlabeled files, forward reading/writing and backwards reading leave the tape positioned at the tape mark separating the files (assuming the LEAVE environment option is used for opening and closing the file). Multiple files may be written forward before reading backwards through some or all of the files.

Due to VSE tape label processing, only one labeled file may be read backwards after a forward read or write. The LEAVE environment option must be used for closing the forward operation and for opening the backwards read.

ENVIRONMENT attribute

The characteristic-list of the ENVIRONMENT attribute specifies various data set characteristics that are not part of the PL/I language. The syntax for the ENVIRONMENT attributes is:



Abbreviation: ENV

characteristic-list

can be any one of the following options:

ADDBUFF[3]	CTL360	LEAVE	SCALARVARYING
ASCII	DSN(n)	LIMCT	SIS[1]
BKWD	EXTENTNUMBER[2]	MEDIUM	SKIP
BLKSIZE(n)	FIFB	NOFEED	TOTAL
BUFFERS(n)	FILESEC	NOLABEL[2]	TRKOFL[1]
BUFND(n)	GENKEY	NOTAPEMK	UIDIDB
BUFNI(n)	GRAPHIC	NOWRITE	UNLOAD
BUFOFF(n)	HIGHINDEX[1]	OFLTRACKS	VIVBIVSIVBS
BUFSP(n)	INDEXAREA(n)[1]	PASSWORD	VERIFY
CMDCHN	INDEXED	RECSIZE(n)	VOLSEQ
COBOL	INDEXMULTIPLE[1]	REGIONAL(1|2|3)	VSAM
CONSECUTIVE	KEYLENGTH(n)	REREAD	WRTPROT
CTLASA	KEYLOC(n)	REUSE	

Note:

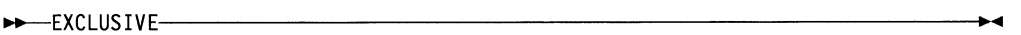
1. Syntax checked only; has no effect. Kept for compatibility with PL/I MVS & VM.
2. Syntax checked only; has no effect. Kept for compatibility with DOS PL/I.
3. Syntax checked only; has no effect. Kept for compatibility with previous releases of PL/I.

Options in the characteristic-list are separated by blanks or commas. The options are described in the *PL/I VSE Programming Guide.*

EXCLUSIVE attribute

Syntax checked only; has no effect. Kept for compatibility with PL/I MVS & VM.

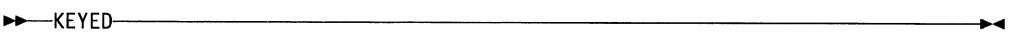
The EXCLUSIVE attribute specifies that records in a file can be locked by an accessing task to prevent other tasks from interfering with an operation. The syntax for the EXCLUSIVE attribute is:



Abbreviation: EXCL

KEYED attribute

The KEYED attribute applies only to RECORD files, and must be associated with direct access devices. It specifies that records in the file can be accessed using one of the key options (KEY, KEYTO, or KEYFROM) of data transmission statements or of the DELETE statement. The syntax for the KEYED attribute is:



The KEYED attribute need not be specified unless one of the key options is used. The nature and use of keys are discussed in detail in the *PL/I VSE Programming Guide.*

PRINT attribute

The PRINT attribute is described under "PRINT attribute" on page 287.

Opening and closing files

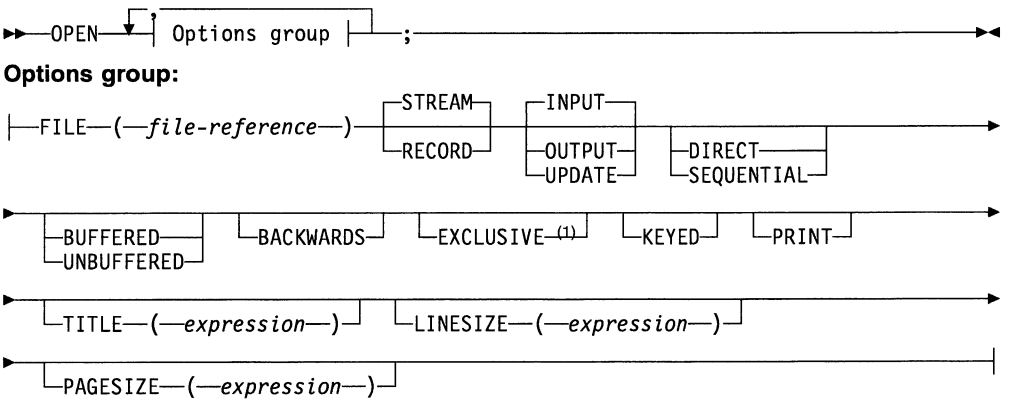
Before a file can be used for data transmission, by input or output statements, the file must be associated with a data set. Opening a file associates a file with a data set and involves checking for the availability of external storage media, positioning the media, and allocating appropriate operating 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, these statements are optional. If an OPEN statement is not executed for a file, the file is opened implicitly before the first data transmission statement for that file is executed. In this case, the file opening uses information about the file as specified in a DECLARE statement (or defaults derived from the transmission statement). Similarly, if a file has not been closed before a program ends, the file is closed during program termination.

When a file for stream input, sequential input, or sequential update is opened, the associated data set is positioned at the first record. When a BACKWARDS file is opened, the associated data set is positioned at the last record.

OPEN statement

The OPEN statement associates a file with a data set (see the *PL/I VSE Programming Guide* for additional information). It also can complete 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:



Note:
[1] Syntax checked only; has no effect. Kept for compatibility with PL/I MVS & VM.

The options of the OPEN statement can appear in any order.

FILE specifies the name of the file that is associated with a data set.

STREAM, RECORD,
INPUT, OUTPUT, UPDATE,
DIRECT, SEQUENTIAL
BUFFERED, UNBUFFERED,
BACKWARDS
EXCLUSIVE,
KEYED,
and PRINT

options specify attributes that augment the attributes specified in the file declaration. The file options can appear in any order. The same attributes need not be listed in both an OPEN statement and a DECLARE statement for the same file, and there must be no conflict.

TITLE is converted to a character string, if necessary. The first 7 characters of the character string identify the data set associated with the file. If the TITLE option does not appear, the default uses the first 7 characters of the file name (padded or truncated). This is not the same truncation as that for external names.

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 files 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, for V-format records).

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 STREAM and PRINT attributes.

When a STREAM file is opened, if the first GET or PUT specifies, by means of a statement option or format item, that n lines are to be skipped before the first record is accessed, the file is then positioned at the start of the nth record. Otherwise, it is positioned at the start of the first line or record. If the file has the PRINT attribute, it is physically positioned at column 1 of that line.

The opening of an already-open file does not affect the file. In such cases, any expressions in the *options-group* are evaluated, but they are not used.

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.

The following list shows the attributes that are implied when an implicit opening is caused by the statement in the left-hand column:

Statement	Implied attributes
GET	STREAM, INPUT
PUT	STREAM, OUTPUT
READ	RECORD, INPUT[1], OUTPUT[1]
WRITE	RECORD, INPUT[1], OUTPUT[1]
LOCATE	RECORD, OUTPUT, SEQUENTIAL, BUFFERED
REWRITE	RECORD, UPDATE
DELETE	RECORD, UPDATE

Notes:

1. INPUT and OUTPUT are default attributes for READ and WRITE statements only if UPDATE has not been explicitly declared.

An implicit opening caused by one of the above statements is equivalent to preceding the statement with an OPEN statement that specifies the same attributes.

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 attribute implications listed below are applied prior to the application of the default attributes discussed earlier. Implied attributes can also cause a conflict. If

a conflict in attributes exists after the application of default attributes, the UNDEFINEDFILE condition is raised.

Following is a list of merged attributes and attributes that each implies after merging:

Merged attributes	Implied attributes
UPDATE	RECORD
SEQUENTIAL	RECORD
DIRECT	RECORD, KEYED
BUFFERED	RECORD
UNBUFFERED	RECORD
PRINT	OUTPUT, STREAM
BACKWARDS	RECORD, SEQUENTIAL, INPUT
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, SEQUENTIAL, and UNBUFFERED.

The following example illustrates implicit opening:

```
DECLARE MASTER FILE KEYED INTERNAL;
     ENVIRONMENT (INDEXED F
     RECSIZE(120) KEYLEN(8));

READ FILE (MASTER)
     INTO (MASTER_RECORD)
     KEYTO(MASTER_KEY);
```

Attributes after merge (due to 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, SEQUENTIAL, and BUFFERED.

Following are examples of declarations of file constants including the ENVIRONMENT attribute:

```
DECLARE FILE#3 INPUT DIRECT
     ENVIRONMENT(V BLKSIZE(328)
     REGIONAL(3));
```

This declaration specifies three file attributes: INPUT, DIRECT, and ENVIRONMENT. Other implied attributes are FILE (implied by any of the attributes) and RECORD and KEYED (implied by DIRECT). Scope is EXTERNAL, by default. The ENVIRONMENT attribute specifies that the data set is of the REGIONAL(3) organization and contains unblocked varying-length records with a maximum length of 328 bytes. A maximum length record will contain only 320 bytes of data used by the program, because 8 bytes are required for control information in V-format records. The KEY option must be specified in each READ statement that refers to this file.

```
DECLARE INVNTRY UPDATE BUFFERED /* FIRST EXAMPLE DECLARE INVNTRY */
     ENVIRONMENT (F RECSIZE(100) VSAM);
```

This declaration also specifies three file attributes: UPDATE, BUFFERED, and ENVIRONMENT. Implied attributes are FILE, RECORD, and SEQUENTIAL (the last two attributes are implied by BUFFERED). Scope is EXTERNAL, by default. The data set is of VSAM organization, and contains fixed-length records of 100 bytes each. Note that, although the data set actually contains recorded keys, the KEYTO option cannot be specified in a READ statement, since the KEYED attribute has not been specified.

For both the above declarations, all necessary attributes are either specified or implied in the DECLARE statement; you cannot change any attributes at run time.

Here is a more flexible declaration for INVNTRY:

```
DECLARE INVNTRY /* SECOND EXAMPLE DECLARE INVNTRY */
     ENVIRONMENT(F RECSIZE(100) VSAM);
```

With this declaration, you can open INVNTRY for different purposes. For example:

```
OPEN FILE (INVNTRY)
     UPDATE SEQUENTIAL BUFFERED;
```

With this OPEN statement, the file attributes are the same as those specified (or implied) in the first example DECLARE INVNTRY. The file might be opened in this way, then closed, and then later opened with a different set of attributes. For example:

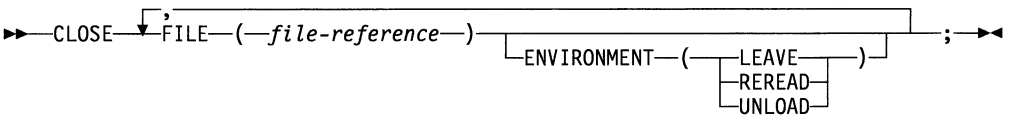
```
OPEN FILE (INVNTRY)
     INPUT SEQUENTIAL KEYED;
```

This OPEN statement allows records to be read with either the KEYTO or the KEY option. Because the file is SEQUENTIAL and the data set is VSAM, the data set can be accessed in a purely sequential manner. It can also be accessed randomly using 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.

CLOSE statement

The CLOSE statement dissociates the file from the data set with which it is associated when the file is opened. The syntax for the CLOSE statement is:



FILE The name of the file that is to be dissociated from the data set.

ENVIRONMENT
Controls disposition of magnetic tapes (for more on this, see the *PL/I VSE Programming Guide).*

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 specified for the file constant in a subsequent OPEN statement. However, all attributes explicitly given to the file constant in a DECLARE statement remain in effect.

Closing an already-closed file has no effect except increasing the run time of the program.

A closed file can be reopened.

If a file is not closed by a CLOSE statement, it is closed at the completion of the program in which it was opened.

All input/output events associated with the file that have a status value of zero when the file is closed are set to complete, with a status value of 1.

Chapter 10. Record-oriented data transmission

Chapter 10. 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 equally to record-oriented and stream-oriented data transmission, including files, file attributes, and opening and closing files, are described in Chapter 9, "Input and output." The ENVIRONMENT attribute and details of record I/O data transmission statements for each data set organization are described in the *PL/I VSE 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 in a buffer. On output, the WRITE, REWRITE, or LOCATE statement transmits a single record from a program variable exactly as it is recorded internally.

Although, for non-VSAM data sets, data is transmitted to and from a data set in blocks, the record-oriented data transmission statements are concerned only with records. The records are blocked and deblocked automatically.

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. There are certain special considerations for a few types of data, and these are given below.

Data aggregates

An aggregate must be in connected storage.

Unaligned bit strings

The following cannot be transmitted:

- BASED, DEFINED, parameter, subscripted, or structure-base-element variables that are unaligned fixed-length bit strings
- Minor structures whose first or last base elements are unaligned fixed-length 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 fixed-length bit strings as their first or last elements

Varying-length strings

A locate mode output statement (see "LOCATE statement" on page 254) specifying a varying-length 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 **must** be specified for the file, otherwise the results will be incorrect.

A move mode output statement (see "WRITE statement" on page 254 and "REWRITE statement" on page 254) specifying a varying-length string variable

transmits only the current length of the string. A 2-byte prefix is included only if the SCALARVARYING option is specified for the file.

Graphic strings

If a graphic string is specified for input or output, the SCALARVARYING option must be specified for the file.

Area variables

A locate mode output statement specifying an area variable transmits a field as long as the declared size of the area, plus a 16-byte prefix containing control information.

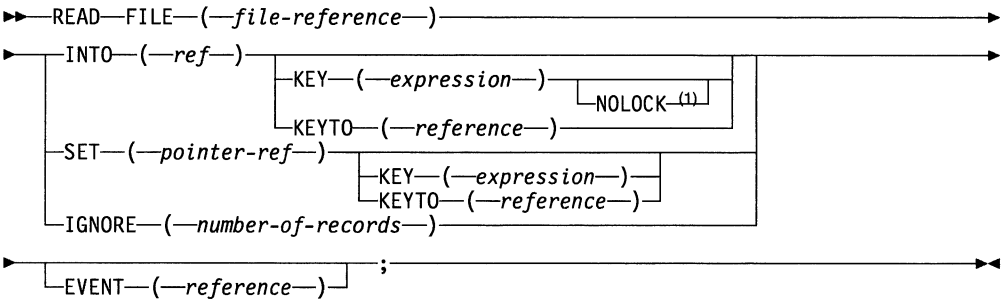
A move mode statement specifying an element area variable or a structure whose last element is an area variable transmits only the current extent of the area plus a 16-byte prefix.

Data transmission statements

The data transmission statements that transmit records to or from auxiliary storage 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. Data transmission statements and options allowed for each data set organization are listed in the *PL/I VSE Programming Guide.*

READ statement

The READ statement can be used with any INPUT or UPDATE file. It transmits a record from the data set to the program, either directly to a variable or to a buffer. In blocked records, a READ statement with the appropriate option transfers a record from a buffer to the variable or sets a pointer to the record in a buffer. Consequently, not every READ statement transmits data from an input device. The syntax for the READ statement is:



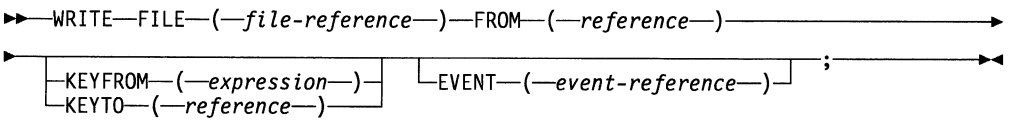
Note:

[1] Syntax checked only; has no effect. Kept for compatibility with PL/I MVS & VM.

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

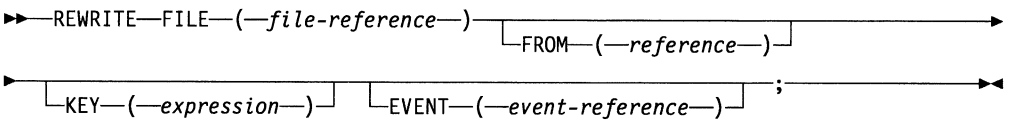
The WRITE statement can be used with any OUTPUT file or DIRECT UPDATE file, and also with SEQUENTIAL UPDATE files associated with VSAM data sets. It transmits a record from the program and adds it to the data set. For unblocked records, transmission can be directly from a variable or from a buffer. For blocked records, the WRITE statement places a logical record into a buffer; only when the blocking of the records is complete is there actual transmission of data to an output device. 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, and for KEYED SEQUENTIAL UPDATE files associated with VSAM data sets, any record can be rewritten whether or not it has first been read. The syntax for the REWRITE statement is:



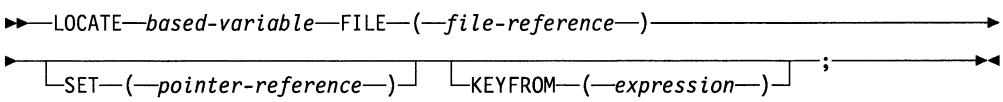
The keywords can appear in any order. The FROM option must be specified for UPDATE files having either the DIRECT attribute or both the SEQUENTIAL and UNBUFFERED attributes.

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 buffer identified by the pointer variable in the SET option. When the records are blocked, a REWRITE statement issued for any record in the block rewrites the complete block even if no REWRITE statements are issued for other records in the block.

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 in the buffer. For further description of locate mode processing, see "Locate mode" on page 262. The syntax for the LOCATE statement is:

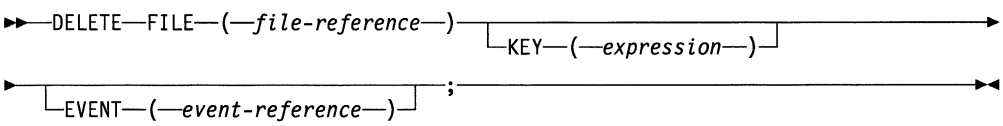


The keywords can appear in any order.

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:

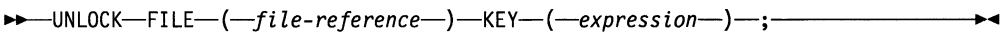


The keywords can appear in any order.

For a SEQUENTIAL UPDATE file of a VSAM KSDS, RRDS, or VRDS: if the KEY option is omitted, the record to be deleted is the last record that was read.

UNLOCK statement

Syntax checked only; has no effect. Kept for compatibility with PL/I MVS & VM. The UNLOCK statement makes the specified locked record available to other MVS tasks. The syntax for the UNLOCK statement is:



The keywords can appear in any order.

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. Lists of the allowed combinations for each type of file are given in the *PL/I VSE Programming Guide.*

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, it is opened implicitly.

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 fixed-length bit strings are disallowed (for details, see "Data transmitted" on page 252).

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-length 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);
```

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 fixed-length bit strings are disallowed (for details, see "Data transmitted" on page 252).

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-length string is transmitted to a data set. If the SCALARVARYING option of the ENVIRONMENT attribute is specified for the file, a 2-byte prefix specifying the string length is attached.

Records are transmitted as an integral number of bytes. Therefore, 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 will contain bits at the start or at the end that are not part of the string.

The FROM option can be omitted from a REWRITE statement for SEQUENTIAL BUFFERED 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 in the buffer (updated by whatever assignments were made) is copied back onto the data set.

In the following example, both statements specify that the value of the variable MAS_REC is written into the file MASTER. The WRITE statement specifies a new

record in a SEQUENTIAL OUTPUT file. The REWRITE statement specifies that MAS_REC replaces the last record read from a SEQUENTIAL UPDATE file.

```
WRITE FILE (MASTER) FROM (MAS_REC);
REWRITE FILE (MASTER) FROM (MAS_REC);
```

SET option

The SET option specifies a pointer variable that is set to point to the location in the buffer into which data has been moved during the READ operation, or which has been allocated by the LOCATE statement. The SET option can be used with a READ statement or a LOCATE statement.

For the LOCATE statement, if the SET option is omitted, the pointer declared with the record variable is set.

If an element string variable of varying-length 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);
```

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, relative to the record pointer prior to IGNORE. 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:

```
READ FILE (IN) IGNORE (3);
```

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 associated with data sets of REGIONAL or VSAM organization. The KEY option must appear if the file has the DIRECT attribute. The KEY option can also appear for a file having VSAM organization and 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 (STOCK) INTO (ITEM) KEY (STKEY);
```

KEYFROM option

The KEYFROM option specifies a character or graphic key that identifies the record on the data set. It can be used in a WRITE statement for a SEQUENTIAL OUTPUT or DIRECT UPDATE file or a DIRECT OUTPUT file that has REGIONAL organization, or in a LOCATE statement. It can also be used in a WRITE statement for a KEYED SEQUENTIAL UPDATE file associated with a VSAM data set.

The KEYFROM option applies only to KEYED files associated with data sets of REGIONAL or VSAM organization. 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.

REGIONAL(1) data sets can be created using the KEYFROM option. The region number is specified as the key.

For REGIONAL(2) and REGIONAL(3) data sets, KEYFROM specifies a recorded key whose length is determined by the KEYLENGTH option.

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 associated with data sets of REGIONAL or VSAM organization.

Assignment to the KEYTO variable always follows assignment to the INTO variable. If an incorrect key specification is detected, the KEY condition is raised. For this implementation, the value assigned is as follows:

- For REGIONAL(1), the 8-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.
- For REGIONAL(2) and REGIONAL(3), the recorded key without the region number, padded or truncated on the *right* to the declared length of the character variable.
- For key-sequenced VSAM data sets (KSDS), the recorded key, padded or truncated on the *right* to the declared length of the character variable.
- For entry-sequenced VSAM data sets (ESDS), a 4-character relative-byte address (RBA), padded or truncated on the *right* to the declared length of the character variable.

- For relative-record VSAM data sets (RRDS) and variable-length relative-record VSAM data sets (VRDS) an 8-character relative-record number with leading zeros suppressed, truncated or padded on the *left* to the declared length of the character variable.

The KEY condition is not raised for this type of padding or truncation.

The KEYTO option can also be used in a WRITE statement for a SEQUENTIAL OUTPUT or SEQUENTIAL UPDATE file associated with a VSAM entry-sequenced or relative-record data set.

The KEYTO option can be used to obtain the relative-byte address (RBA) when a record is added to a VSAM entry-sequenced data set, or the relative-record number when a record is added to a VSAM relative-record data set. The character value returned for an ESDS is of length 4 representing an RBA. The character value returned for an RRDS or VRDS is of length 8, representing an unsigned decimal integer value with leading zeros suppressed.

The following example specifies that the next record in the file DETAIL is read into the variable INVTRY, and that the key of the record is read into the variable KEYFLD:

```
READ FILE (DETAIL) INTO (INVTRY) KEYTO (KEYFLD);
```

EVENT option

The EVENT option specifies that the input or output operation takes place asynchronously (that is, while other processing continues) and that no I/O conditions (except for UNDEFINEDFILE) are raised until a WAIT statement, specifying the same event variable, is executed.

The following example shows how to use the EVENT option:

```
READ FILE (MASTER) INTO (REC_VAR)
     EVENT (RECORD_1);
        .
        .
        .
WAIT (RECORD_1);
```

The EVENT option can appear in any READ, WRITE, REWRITE, or DELETE statement for an UNBUFFERED file with CONSECUTIVE or REGIONAL organization or for any DIRECT file.

A name declared implicitly that appears in an EVENT option is given the EVENT attribute.

When any expressions in the options of the statement have been evaluated, the input operation is started, and the event variable is made active (that is, the variable cannot be associated with another event) and is given the completion value '0'B and zero status value, provided that the UNDEFINEDFILE condition is not raised by an implicit file opening (see "Note" below). The sequence of these two assignments is uninterruptible and is completed before any transmission is initiated but after any action associated with an implicit opening is completed.

As soon as this has happened, the statements following are executed. Any RECORD, TRANSMIT, KEY, or ENDFILE condition is not raised until control

reaches the WAIT statement. The event variable remains active and retains its '0'B completion value until control reaches a WAIT statement specifying that event variable, or until termination of the application program.

When the WAIT statement is executed, any of the following can occur:

- If the input/output operation is not complete, and if none of the four conditions is raised, execution of further statements is suspended until the operation is complete.
- If the input/output operation is executed successfully and none of the conditions ENDFILE, TRANSMIT, KEY or RECORD is raised as a result of the operation, the event variable is set complete (given the completion value '1'B), and the event variable is made inactive (that is, it can be associated with another event).
- Any ENDFILE, TRANSMIT, KEY, or RECORD conditions for the input/output operation are raised when the WAIT is encountered. At such a time, the event variable is set to have a status value of 1 and the corresponding ON-units (if any) are entered in the order in which the conditions were raised. After a return from the final ON-unit, or if one of the ON-units is terminated by a GO TO statement (abnormal return), the event variable is given the completion value '1'B and is made inactive.

If some of the event variables in the WAIT list are associated with input/output operations and have not been set complete before the WAIT is terminated (either because enough events have been completed or due to an abnormal return), these incomplete events are not set complete until the execution of another WAIT referring to these events.

Note: If the statement causes an implicit file opening that results in the raising of the UNDEFINEDFILE condition, the ON-unit associated with this condition is entered immediately and the event variable remains inactive and retains the same value it had when the statement was encountered. If the ON-unit does not correct the condition, then, upon normal return from the ON-unit, the ERROR condition is raised. If the condition is corrected in the ON-unit, that is, if the file is opened successfully, upon normal return from the ON-unit, the event variable is set to '0'B, it is made active, and execution of the statement continues.

Upon normal return from any ON-units entered, processing continues with the next statement following the WAIT statement.

For consecutive and regional sequential files, only one outstanding input/output operation is allowed for a file. The ERROR condition is raised if an attempt is made to initiate an input/output operation on a file in excess of the number allowed, while a previous input/output operation has not been waited for.

NOLOCK option

Syntax checked only; has no effect. Kept for compatibility with PL/I MVS & VM.

Processing modes

Record-oriented data transmission has two modes of handling data:

Move mode You can process data by having the data moved into or out of the variable, either directly or via a buffer.

Locate mode You can process data while it remains in a buffer.

The execution of a data transmission statement assigns to a pointer variable the location of the storage allocated to a record in the buffer. Locate mode is applicable only to BUFFERED files. The file must be either a SEQUENTIAL file or an INPUT or UPDATE file associated with a VSAM data set.

Which mode is used is determined by the data transmission statements and options that you use.

For VSAM data sets, PL/I uses locate mode only when there is no data transferred to the application (for example, for READ with the IGNORE option). The PL/I LOCATE statement is supported by move mode I/O using internal data buffers.

Move mode

In move mode, a READ statement transfers a record from external storage to the variable named in the INTO option (via an input buffer if a BUFFERED file is used). A WRITE or REWRITE statement transfers a record from the variable named in the FROM option to external storage (perhaps via an output buffer). The variables named in the INTO and FROM options can be of any storage class.

Move mode can result in faster execution when there are numerous references to the contents of the same record, because of the overhead incurred by the indirect addressing technique used in locate mode.

It is possible to use the move mode access technique and avoid internal movement of data in the following cases:

- SEQUENTIAL UNBUFFERED files with: CONSECUTIVE organization with either U-format records, or F-format records that are not larger than the variable specified in either the INTO or FROM option; and REGIONAL(1) organization with F-format records that are not larger than the variable specified in the FROM or INTO option.
- DIRECT files with REGIONAL(1) or REGIONAL(2) organization and F-format records, and REGIONAL(3) organization with F-format or U-format records.

Consider the following example, illustrated in Figure 11:

```
      ON ENDFILE(IN) GO TO EOF_IN;
NEXT: READ FILE(IN) INTO(DATA);
        .
        .
        .
      GO TO NEXT;
EOF_IN:;
```

The first time the READ statement is executed, a block is transmitted from the data set associated with the file IN to an input buffer, and the first record in the block is assigned to the variable DATA. Further executions of the READ statement assign

successive records from the buffer to DATA. When all the records in the buffer have been processed, the next READ statement transmits a new block from the data set, although this READ statement will probably access a new record in an alternative buffer. Use of multiple buffers and the move mode allows overlapped data transmission and processing.

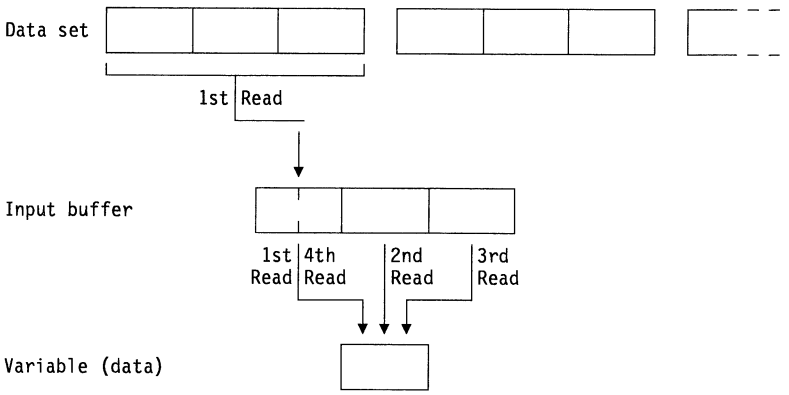


Figure 11. Move mode input

A move mode WRITE statement is executed in a similar manner, building physical records in an output buffer and transmitting them to the data set associated with the file each time the buffer is filled.

Locate mode

Locate mode assigns to a pointer variable the location of an input or output buffer. A based variable provides the attributes of the data in the buffer. 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 locate-mode READ statement (a READ statement with a SET option) transfers a block of data from the data set to an input buffer if necessary, and then sets the pointer variable in the SET option to point to the location in the buffer of the next 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 LOCATE statement is used for output from a buffer for SEQUENTIAL files. A LOCATE statement allocates storage within an output buffer for a based variable and does the following:

1. Transfers a block of data to the data set from an output buffer, if the current block is complete.
2. Sets a pointer variable to the location in the buffer of the next output record. 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.

3. Initializes components of the based variable that have been specified in REFER options.

After execution of the LOCATE statement, values can be assigned directly into the output buffer by referencing based variables qualified by the pointer variable set by the LOCATE statement. If the current block is complete, the next LOCATE, WRITE, or CLOSE statement for the same file transmits the data in the output buffer to the data set.

Locate mode can result in faster execution than move mode since there is less movement of data, and less storage may be required.

Figure 12 illustrates the following example, which uses locate mode for input:

```
     DCL DATA BASED(P);
     ON ENDFILE(IN) GO TO EOF_IN;
NEXT:  READ FILE(IN) SET(P);
         .
         .
         .
     GO TO NEXT;
EOF_IN:;
```

The first time the READ statement is executed, a block is transmitted from the data set associated with the file IN to an input buffer, and the pointer variable P is set to point to the first record in the buffer. Any reference to the variable DATA or to any other based variable qualified by the pointer P is then a reference to this first record. Further executions of the READ statement set the pointer variable P to point to succeeding records in the buffer. When all the records in the buffer have been processed, the next READ statement transmits a new block from the data set.

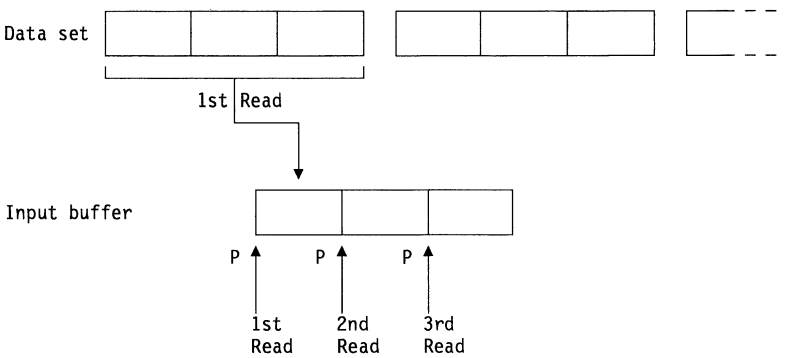


Figure 12. Locate mode input

Locate mode output is shown in the following example:

```
     DCL DATA BASED(P);
NEXT: LOCATE DATA FILE(OUT);
     DATA = ...;
     GO TO NEXT;
```

Each execution of the LOCATE statement reserves storage in an output buffer for a new allocation of the based variable DATA and sets the pointer variable P to point to this storage. When no more space is available in the output buffer, the next execution of the LOCATE statement transmits a block to the data set associated with the file OUT, and allocates a new buffer.

Record alignment

When using locate mode input/output, the first data byte of the first record in a block is generally aligned in a buffer on a doubleword boundary (see Figure 16). The next record begins at the next available byte in the buffer. If the alignment of this byte matches the alignment requirements of the based variable with which the record is associated, it can result in better processor performance.

For blocked records, doubleword alignment of the first byte of data in each record in the block is ensured if the record length (RECSIZE) is a multiple of 8. For spanned records, the block size (BLKSIZE) must be a multiple of 8 if this alignment is required. For data read from ASCII data sets, the first byte of the block prefix is doubleword-aligned. To ensure similar alignment of the first byte of the first record, the prefix length must be a multiple of 8 bytes, less 4 to allow for the 4 record-length bytes.

Most of the alignment requirements described here occur in ALIGNED based or nonbased variables. If these variables are UNALIGNED, the preservation of the record alignment in the buffer is considerably easier.

If a VB-format record is constructed with logical records defined by the structure:

```
1 S,
  2 A CHAR(1),
  2 B FIXED BINARY(31,0);
```

this structure is mapped as in Figure 13.

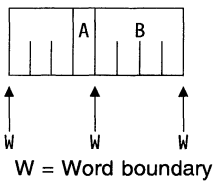


Figure 13. Format of structure S

If the block is created using a sequence of WRITE FROM(S) statements, the format of the block is like Figure 14. You can see that the alignment in the buffer differs from the alignment of S.

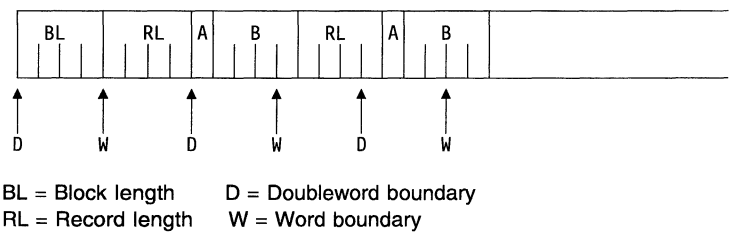


Figure 14. Block created from structure S

Alignment is correct if the file is then read using move mode READ statements, for example, READ INTO(S), because information is moved from the buffer to correctly aligned storage. However, if a structure is defined as:

```
1 SBASED BASED(P) LIKE S;
```

and READ SET(P) statements are used, some references to SBASED.B will not be correctly aligned, as the SET option of the READ statement sets P to the address of the buffer.

The same incorrect alignment occurs if the file is created by using the statement:

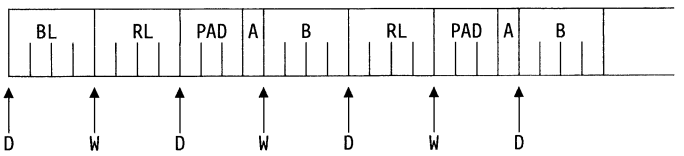
```
LOCATE SBASED SET(P);
```

Again, for the first record in the block, P is set to address a doubleword, and references to SBASED.B do not align correctly.

In both cases you can pad the structure in such a way that B is always correctly aligned:

```
1 S,
  2 PAD CHAR(3),
  2 A CHAR(1),
  2 B FIXED BINARY(31,0);
```

The block format is like Figure 15. B is always on a word boundary. Padding might be required at the beginning and end of a structure to preserve alignment.



BL = Block length D = Doubleword boundary
RL = Record length W = Word boundary

Figure 15. Block created by structure S with correct alignment

The alignment of different types of records within a buffer is shown in Figure 16. The first data byte in a block (or hidden buffer) is always on a doubleword boundary. The position of any successive records in the buffer depends on the record format.

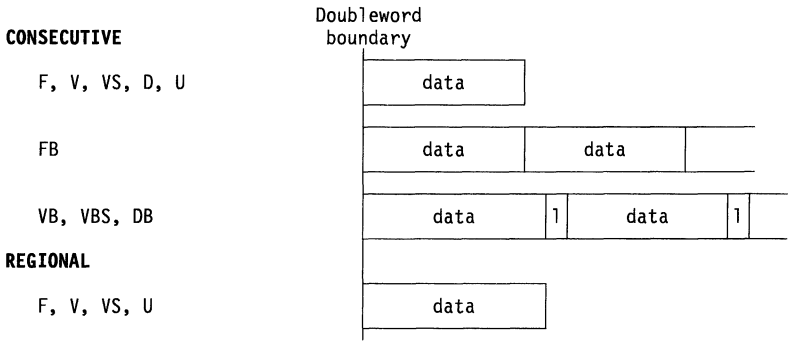
When using locate mode input/output in conjunction with a based variable containing a REFER option, you need to consider alignment requirements when determining an adjustable extent, although sometimes no action is required. Consider the following structure:

```
1 S BASED(P),
  2 N,
  2 C CHAR (L REFER (N));
```

If you want to create blocked V-format records of this type with correct record alignment, using locate mode input/output, record alignment must be such that N is half-word aligned. If L is not a multiple of 2 then, if the alignment of the current record is correct, that of the following record will be incorrect. Correct alignment can be obtained by the following sequence:

```
LENGTH = L;
  /* SAVE DESIRED LENGTH L */
L = 2* CEIL(L/2);
  /* ROUND UP TO MULTIPLE OF 2*/
LOCATE S FILE (F);
N = LENGTH;
  /* SET REFER VARIABLE */
```

This technique can be adapted to other uses of the REFER option.



Notes:

1. l = record length
2. Each I/O operation sets the pointer to the beginning of the data in the records.
3. For CONSECUTIVE data sets with VBS format records, if the record length is greater than the block size, the record is moved to a hidden buffer, with the first data byte on a doubleword boundary.

Figure 16. Alignment of data in a buffer in locate mode input/output

Chapter 11. Stream-oriented data transmission

Chapter 11. Stream-oriented data transmission

This chapter describes the input and output statements used in stream-oriented data transmission. Those features that apply to stream-oriented and record-oriented data transmission, including files, file attributes, and opening and closing files, are described in Chapter 9, "Input and output" on page 236. Other related information is in the *PL/I VSE Programming Guide*.

Stream-oriented data transmission treats a data set as a continuous stream of data values in character, graphic, or mixed data form. Within a program, block and record boundaries are 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; however, 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

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

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. Data-directed output is useful for producing annotated output.

Edit-directed data transmission

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.

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. Continuation rules are applied in such files, and are the same rules as those described in "DBCS continuation rules" on page 16. Any invalid use of shift codes (the rules for graphic and mixed constants apply) raises the ERROR condition. For information on how graphics are handled for edit-directed transmission, see the discussion under "Edit-directed data specification" on page 284.

Data transmission statements

Stream-oriented data transmission uses only one input statement, GET, and one output statement, PUT. The FORMAT statement specifies the format of data values being transmitted and is used only for edit-directed data transmission.

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.

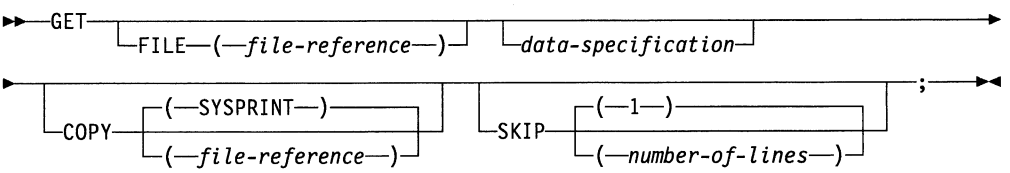
Only sequential files can be processed with the GET and PUT statements.

GET statement

The GET statement is a STREAM input data transmission statement that can either:

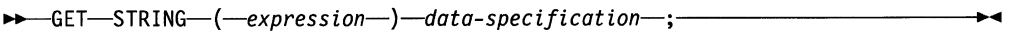
- Assign data values from a data set to one or more variables
- Assign data values from a string to one or more variables

The syntax for 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 for the GET statement for transmission from a string is:

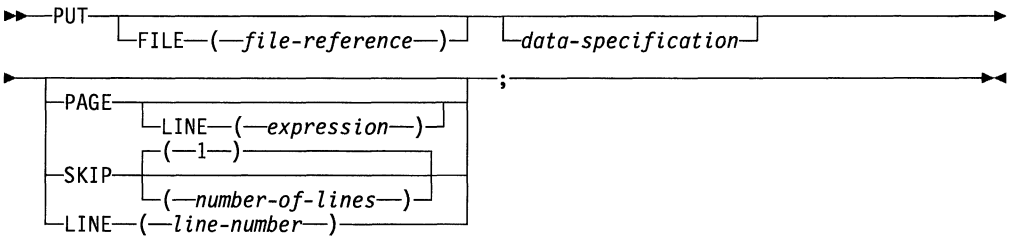


PUT statement

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 for the PUT statement for a stream output file is:

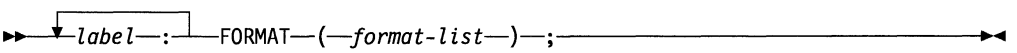


The data specification can be omitted only if one of the control options (PAGE, SKIP, or LINE) appears.

The syntax for the PUT statement for transmission to a character string is:

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 The value of the label-reference of a remote format item must be the label constant of a FORMAT statement.

format-list
Described under "Edit-directed data specification" on page 284.

A GET or PUT statement can include a remote format item, R, in the format-list of an edit-directed data specification. That portion of the format-list represented by the R format item is supplied by the identified FORMAT statement. The remote format item and the FORMAT statement must be internal to the same invocation of the same block. For a description of the R-format item, see "R-format item" on page 300.

If a condition prefix is associated with a FORMAT statement, it must be identical to the condition prefix associated with the GET or PUT statement referring to that FORMAT statement.

When a FORMAT statement is encountered in normal sequential flow, control passes around it.

Options of data transmission statements

Options that you can specify on stream-oriented data transmission statements are as follows:

FILE option

The FILE option specifies the file upon which the operation takes place. It must be a STREAM file.

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.

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. If they are written by default on the SYSPRINT file, they appear in data-directed format. 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, the file is implicitly opened for stream output transmission.

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 *number-of-lines* 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 output 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 output 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 244.

If fewer than *n* lines remain on the current page when a SKIP(n) is issued, ENDPAGE is raised.

PAGE option

The PAGE option can be specified only for output 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). When a PAGE format item is encountered, a new page is defined.

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, which may result in the definition of a new page. A new current page implies line one. For information on the ENDPAGE condition, see "ENDPAGE condition" on page 336.

LINE option

The LINE option can be specified only for output 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 this case, the effect is as for a SKIP(0) option.

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 244.

STRING option

The STRING option in GET and PUT statements transmits data between main storage locations rather than between the main and auxiliary storage facilities. 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.

In the STRING option of a PUT statement, the *character-reference* cannot be the STRING pseudovariable.

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 usual. 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 STRING option is most useful with edit-directed transmission. The COLUMN control format option cannot be used with the STRING option.

The STRING option 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). The character that is ignored in the second GET statement is the same character that is assigned to CODE by the first GET statement.

```
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.

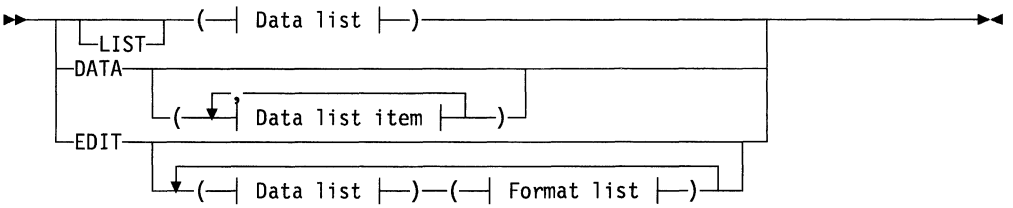
The variable referenced in the STRING option should not be referenced by name or by alias in the data list. For example:

```
DECLARE S CHAR(8) INIT('YYMMDD');
PUT STRING (S) EDIT
    (SUBSTR (S, 3, 2), '/',
    SUBSTR (S, 5, 2), '/',
    SUBSTR (S, 1, 2))
    (A);
```

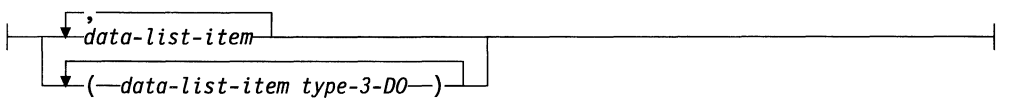
The value of S after the PUT statement is 'MM/bb/MM' and not 'MM/DD/YY' because S is blanked after the first data item is transmitted. The same effect would also be obtained if the data list contained a variable based or defined on the variable specified in the STRING option.

Data specifications

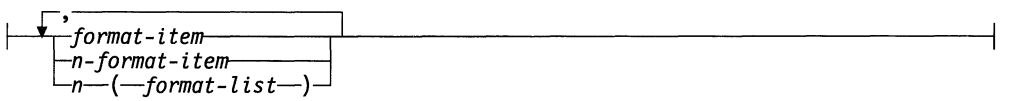
Data specifications in GET and PUT statements specify the data to be transmitted. The syntax for a data specification is:



Data list:



Format list:



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. In such a statement, the data list must immediately follow the GET or PUT keyword; any options required must be specified after the data list.

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 or a pseudovariable other than STRING. 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, locator-qualified, or iSUB-defined. However, qualified (that is, structure-member), simple-defined, 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 or iSUB-defined. It can be qualified (that is, a member

of a structure) or simple-defined or string-overlay-defined. Subscripts are allowed for data-directed output.

The data types of a data-list-item can be:

Input:

Problem data: Coded arithmetic
String

Output:

Problem data: Coded arithmetic
String

Program control data:

Area	Label variable
Entry variable	Offset
Event	Pointer
File	

For list- and data-directed transmission of DBCS data or mixed data, the GRAPHIC option of the ENVIRONMENT attribute must be specified.

A data list that specifies program-control data can only be used in PUT DATA statements that are processed at compile-time. 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 type3-DO

The syntax for the Type 3 do-group is described under "DO statement" on page 175.

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 284.

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:

```
DECLARE 1 A (10), 2 B, 2 C;
PUT FILE(X) LIST(A);
```

results 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, within a data list used in an input statement for list-directed or edit-directed transmission, a variable is assigned a value, this new 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));
```

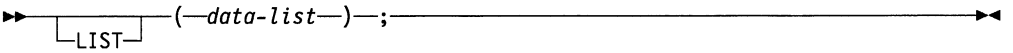
When this statement is executed, values are transmitted and assigned in the following order:

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.
5. A substring of length K is assigned to the string variable NAME, beginning at the Jth character.

List-directed data specification

The syntax for a list-directed data specification is:



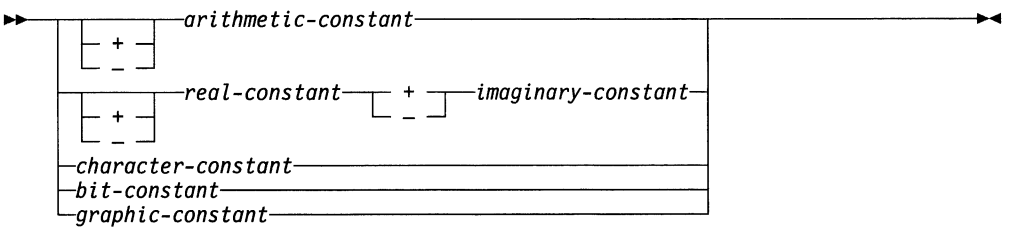
Examples of list-directed data specifications are:

```
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.

List-directed data values

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 + or - 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 4, "Data conversion" on page 88.

GET list-directed

On input, data values in the stream must be separated either by a blank or by a comma. This separator can be surrounded by an arbitrary number of 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 the 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 the record is encountered, in which case the file is positioned at the end of the record:

```
Xbb-bbXXX
  ↑
```

However, if the end of the 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 succeeding record is not read in 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 data is adjusted so that the DBCS portions are enclosed in shift codes.

If the data is a graphic constant, the enclosing shift codes, quotation marks, and the 'G' are removed, and the enclosed graphics are interpreted as a graphic string.

DBCS note: Double quotation marks (a pair of SBCS or a pair of DBCS) are treated as single quotation marks only if the enclosing quotation marks are from the same character set. Single quotation marks are treated as single when the enclosing quotes are not from the same character set.

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

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 287.)

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.

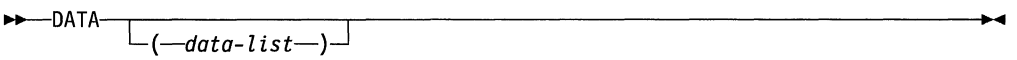
Mixed strings are enclosed in SBCS quotation marks and followed by the letter M. Contained SBCS quotes are replaced by two quotes.

Graphic strings are written out as follows:

- If the file does not have the attribute PRINT, enclosing shift codes, SBCS quotation marks, and the letter G are supplied. Since 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, only enclosing shift codes are supplied. Graphic quotation marks are represented by a single graphic quotation mark (unmodified).

Data-directed data specification

The syntax for a data-directed data specification is:



Names of structure elements in the data-list 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 the variables (except iSUB-defined variables) that are known to the block and any containing blocks.

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.

References to based variables in a data-list for data-directed input/output cannot be explicitly locator qualified. For example:

```
DCL Y BASED(Q), Z BASED;
PUT DATA(Y);
```

The variable Z cannot be transmitted since it must be explicitly qualified by a locator.

The following restrictions apply to based variables in the data-list:

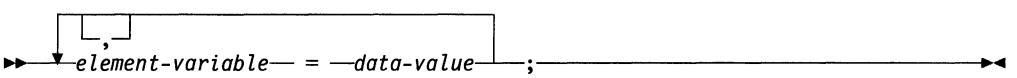
- The variable must not be based on an OFFSET variable.
- The variable must not be a member of a structure declared with the REFER option.
- The pointer on which the variable is based must not be based, defined, or a parameter, and it must not be a member of an array or structure.

Defined variables in the data-list must not be defined:

- On a controlled variable
- On an array with one or more adjustable bounds
- With a POSITION attribute that specifies other than a constant

Data-directed element assignments

The stream associated with data-directed data transmission is in the form of a list of element assignments. For problem data, the element assignments have the following syntax (the optionally signed constants, like the variable names and the equal signs, are in character or graphic form):



The element-variable can be a subscripted name. Subscripts must be optionally-signed integers.

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.

The length of the data value in the stream is a function of the attributes declared for the variable and, since 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 4, "Data conversion."

Qualified names in the input stream must be fully qualified. The name must not contain more than 256 characters.

Locator qualifiers cannot appear in the stream. The locator qualifier declared with the based variable is used to establish the generation. Based variables that are not declared with a locator qualifier cannot be transmitted.

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

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.

Recognition of a semicolon (not enclosed in quotation marks) or an end-of-file causes transmission to cease, and thereby 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';
```

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 CARDIN, 2 PARTNO, 2 DESCRP,
    2 PRICE, 3 RETAIL, 3 WHSL;
```

If it is desired to read a value for CARDIN.PRICE.RETAIL, the input data stream must have the following form:

```
CARDIN.PRICE.RETAIL=1.23;
```

The data specification can be any of:

```
DATA(CARDIN)
DATA(PRICE)
DATA(CARDIN.PRICE)
DATA(RETAIL)
DATA(PRICE.RETAIL)
DATA(CARDIN.RETAIL)
DATA(CARDIN.PRICE.RETAIL)
```

PUT data-directed

A data-list-item can be an element, array, or structure variable, or a repetitive specification. Subscripted names can appear. For problem data, 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 287.)

A semicolon is written into the stream after the last data item transmitted by each PUT statement.

Subscript expressions that appear in a data-list are evaluated and replaced by their values.

Items that are part of a structure appearing in the data-list are transmitted with the full qualification. Subscripts follow the qualified names rather than being interleaved. For example, if a data-list-item is specified for a structure element as follows:

```
DATA (Y(1,-3).Q)
```

the output stream written is:

```
Y.Q(1,-3)= 3.756;
```

Names are transmitted as all SBCS or all DBCS, regardless of how they are specified in the data-list. If a name contains a non-EBCDIC DBCS character, it is transmitted as all DBCS. Each name in a qualified reference is handled independently. For example, if you declared the following structure:

```
DCL 1 ABC,
      2 S<KK> ;
```

the statement

```
PUT DATA (<.A.B.C>.S<kk>);
```

would transmit the following to the output stream:

```
ABC.<.Skk>=...
```

In the following cases, 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.
- When a program control variable is transmitted, the variable must not be specified in an input data list.

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.

Examples

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;
```

For example:

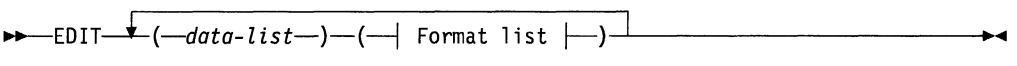
```
DCL 1 A, 2 B FIXED, 2 C, 3 D FIXED;
A.B = 2;
A.D = 17;
PUT DATA (A);
```

the data fields in the output stream are:

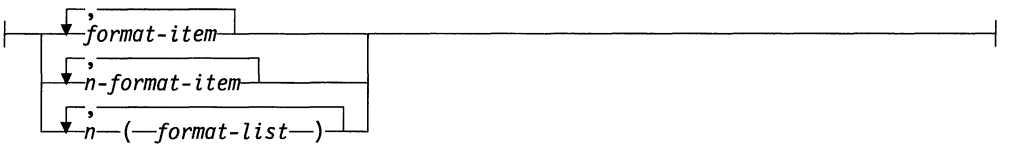
```
A.B= 2 A.C.D= 17;
```

Edit-directed data specification

The syntax for an edit-directed data specification is:



Format list:



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 12, "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.

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 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 244.

A value read into a variable can be used in a format item 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

Data in the stream is a continuous string of characters and graphics without any delimiters between successive values. For files with the GRAPHIC attribute, graphic data must be enclosed in shift codes. 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 (character or graphic) 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 that the first N characters in the stream are treated as a character string and assigned to NAME. The next 2 characters are skipped. The next 6 are assigned to DATA in character format. The next 6 characters are considered an optionally signed decimal fixed-point constant and assigned to SALARY.

PUT edit-directed

The value of each data list item is converted to the character or graphic representation specified by the associated 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:

```
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 5 characters (leading characters can be blanks).

The following examples show 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));
```

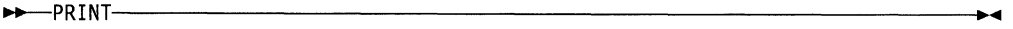
This PUT statement specifies that the heading QUARTERLY STATEMENT is written on line two of a new page in the output file SYSPRINT, two lines are skipped (that is, "skip to the third following line") and the value of ACCT# is written, beginning at the first character of the fifth line; the value of BOUGHT, beginning at character position 14; the value of SOLD, beginning at character position 30; the value of PAYMENT, beginning at character position 45; and the value of BALANCE at character position 60.

```
PUT EDIT (NAME,NUMBER,CITY) (A(N),A(N-4),A(10));
```

In this example, the value of NAME is inserted in the stream as a character string left-adjusted in a field of N characters; NUMBER is left-adjusted in a field of N-4 characters; and CITY is left-adjusted in a field of 10 characters.

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 the compiler.

Data values transmitted by list- and data-directed data transmission are automatically aligned on the left margin and on implementation-defined preset tab positions. These tab positions are 25, 49, 73, 97, and 121, but provision is made for you to alter these values (see the *PL/I VSE Programming Guide).*

The layout of a PRINT file can be controlled by the use of the options and format items listed in Table 23.

Table 23. 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
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.

RECORD files can also be printed. For information, see the *PL/I VSE Programming Guide.*

SYSPRINT file

The file SYSPRINT, unless it is declared explicitly, is given the attribute PRINT. A new page is initiated automatically when the file is opened. If the first PUT statement that refers to the file has the PAGE option, or if the first PUT statement includes a format list with PAGE as the first item, a blank page appears.

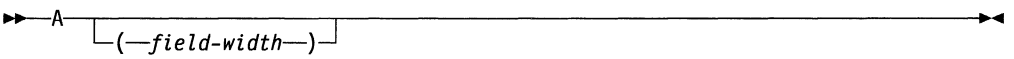
Chapter 12. Edit-directed format items

Chapter 12. 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 284). The format items are described in alphabetic order.

A-format item

The character (or A) format item describes the representation of a character value. The syntax for A format is:



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:

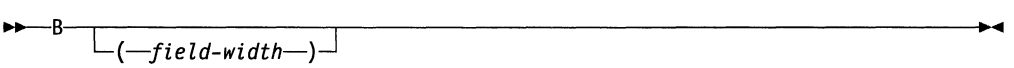
```
GET FILE (INFILE)
EDIT (ITEM) (A(20));
```

This statement assigns the next 20 characters in the file called 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 4, "Data conversion" on page 88).

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. The syntax for B format is:



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 4, "Data conversion" on page 88).

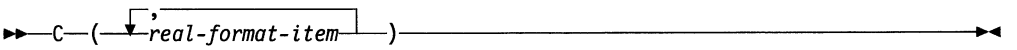
In the example:

```
DECLARE MASK BIT(25);
PUT FILE(MASKFLE) EDIT (MASK) (B);
```

The PUT statement writes the value of MASK in the file called MASKFLE as a string of 25 characters consisting of 0's and 1's.

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. The syntax for C format is:



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.

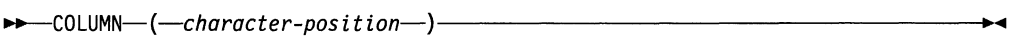
On output, the letter I is never appended to the imaginary part. If the second real format item (or the first, if only one appears) is an F or E item, the sign is transmitted only if the value of the imaginary part is less than zero. If the real format item is a P item, the sign is transmitted only if 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. The syntax for COLUMN format is:



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.

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. In the case of V-format records, this line size is equal to the logical record length *minus* the number of bytes containing control information.

COLUMN must not be used in a GET STRING or PUT STRING statement.

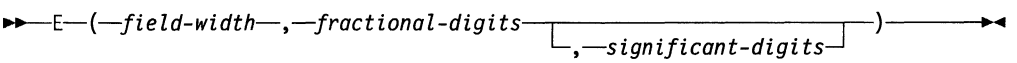
COLUMN cannot be used with input or output lines that contain graphics.

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. The syntax for E format is:



field-width
fractional-digits
significant-digits
Expressions, which are evaluated and converted to integer values (w, d, and s, respectively) each time the format item is used.

Field-width specifies the total number of characters in the field.
Fractional-digits specifies the number of digits in the mantissa that follow

the decimal point. *Significant-digits* specifies the number of digits that must appear in the mantissa.

The following must be true:

```
w >= s >= d  or w = 0
```

and, when w¬= 0

```
s > 0, d >=0
```

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.*

Significant-digits, if it appears, is evaluated and ignored.

If *field-width* is 0, there is no assignment to the data list item.

In the example:

```
GET FILE(A) EDIT (COST) (E(10,6));
```

This statement obtains the next 10 characters from the file called 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.

If *significant-digits* is not specified, it defaults to 1 plus *fractional-digits.*

The character string written in the stream for output has one of the following syntaxes:

- For d=0

  ```
  [-] {s digits}E{±}exponent
  ```

  w must be >=s+4 for positive values, or >=s+5 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

  ```
  [-]{s-d digits}.{d digits}
      E{±}exponent
  ```

  w must be >=s+5 for positive values, or >=s+6 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

  ```
  [-]0.{d digits}E{±}exponent
  ```

  w must be >=d+6 for positive values, or >=d+7 for negative values.

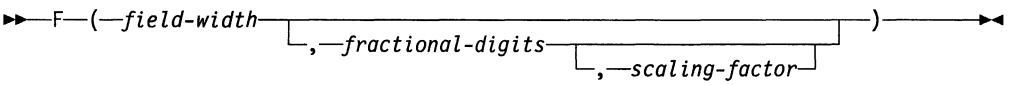
  When the value is nonzero, the exponent is adjusted so that the first fractional digit is nonzero. When the value is zero, each digit position contains zero.

The exponent is a 2-digit integer, which can be two 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. The syntax for F format is:



field-width
fractional-digits
scaling-factor
: Expressions, which are evaluated and converted to integer values (w, d, and p, respectively) each time the format item is used. The evaluated *field-width* and *fractional-digits* must both be nonnegative.

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. (This is different from CHAR to ARITH conversion and from E-format items).

Fractional-digits, if not specified, defaults to 0.

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 (15,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, but *scaling-factor* is not, 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.

The integer value (*p*) of the *scaling-factor* multiplies the value of the data list item, after any conversion to FIXED DECIMAL by 10 raised to the power of *p*, before it is edited into its character representation. When *fractional-digits* is 0, only the integer portion of the data list item is used in this multiplication.

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:

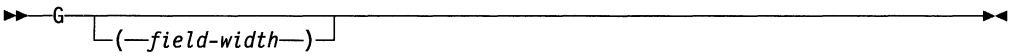
```
DECLARE TOTAL FIXED(4,2);
PUT EDIT (TOTAL) (F(6,2));
```

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. The syntax for G format is:



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. The maximum number that you can specify is 16,383. End of record 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. If the input file has the GRAPHIC option of the ENVIRONMENT attribute, the specified number of graphics must be enclosed in shift codes. 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 graphic quotation marks are inserted, nor is the identifying graphic *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.

If the output file has the GRAPHIC option of the ENVIRONMENT attribute, the specified number of graphics is enclosed in shift codes.

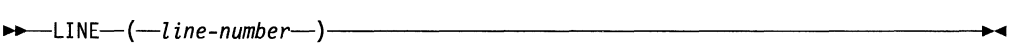
In the example:

```
DECLARE A GRAPHIC(3);
PUT FILE(OUT) EDIT (A) (G(3));
```

If OUT does not have the GRAPHIC option, 6 bytes are transmitted; otherwise, eight bytes (six and the left and right delimiters) are transmitted.

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. The syntax for LINE format is:



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 is treated as 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. The syntax for P format is:

picture-specification
Discussed in detail in Chapter 13, "Picture specification characters."

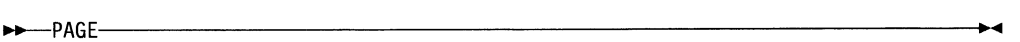
In the example:

```
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. The syntax for PAGE format is:

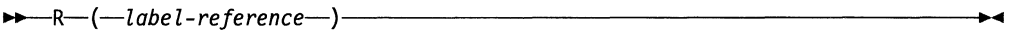


The establishment of a new page positions the file to line one of the next page.

When a PAGE format item is encountered, a new page is defined.

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 270). The syntax for R format is:



label-reference
The value of a FORMAT statement label constant.

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.

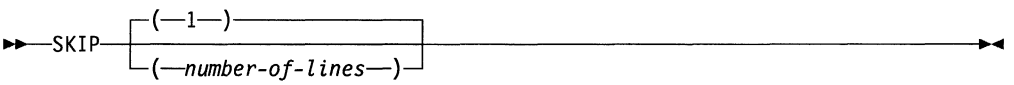
For example:

```
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 to be 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. The syntax for SKIP format is:



number-of-lines
Specifies an expression, which is evaluated and converted to an integer value, n, which must be nonnegative and less than 32,768, each time the format item is used. 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.

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 new 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.

In the 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. The syntax for X format is:

field-width
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.

In the 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 spaced over and 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 names OUT.

Chapter 13. Picture specification characters

Chapter 13. Picture specification characters

A picture specification consists of a sequence of picture characters enclosed in quotation marks, which is either part of the PICTURE attribute (described in "PICTURE attribute" on page 31), or part of the P-format item (described in "P-format item" on page 299) 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, and any other EBCDIC codes.

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 factors

A picture repetition factor specifies the number of repetitions of the immediately-following picture character. A picture repetition factor is an integer, n, enclosed in parentheses. No blanks are allowed within the parentheses. If n is 0, the picture character is ignored. For example, the following picture specification results in the same description:

```
'999V99'
'(3)9V(2)9'
```

Picture characters for character data

A character picture specification describes a fixed-length character data item, with the additional facility of indicating that any position in the data item can only contain characters from certain subsets of the complete set of available characters.

A character picture specification is recognized by the occurrence of an A or X picture specification character. 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 character, #, @, $, or blank.

9 Any digit, or blank. (Note that the *9* picture specification character in numeric character specifications is different in that the corresponding character can only be a digit.)

When a character value is assigned, or transferred, to a pictured character data item, the particular character in each position is checked for validity, as specified by the corresponding picture specification character, and the CONVERSION condition is raised for an invalid character. (However, if you change the value either via record-oriented transmission or 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;'
```

Table 24 shows examples of character picture specifications.

Table 24. 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; therefore, the associated picture specification cannot contain the characters X or A. The picture characters for numeric character data can specify editing of the data.

A numeric character variable can be considered to have two different kinds of value, depending upon its use: an arithmetic value and a character value.

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 (including expressions with the ¬, &, |, and comparison operators; even comparison with a character string uses 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

In such cases, the arithmetic value of the numeric character variable is converted to internal coded arithmetic representation.

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

In such cases, no data conversion is necessary.

The picture characters for numeric character specifications can be grouped into the following categories:

- Digit and decimal-point characters
  9 V
- Zero suppression characters
  Z *
- Insertion characters
  , . / B
- Signs and currency character
  S + - $
- Credit, debit, overpunched, and zero replacement characters
  CR DB T I R Y
- Exponent specifiers
  K E
- Scaling factor
  F

All characters except K, V, and F specify the occurrence of a character in the character representation.

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 portion 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.

Digit and decimal-point characters

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. (The definition of *9* in a character picture is different in that the corresponding character can be blank or a digit).

A string of n *9* picture characters specifies that the item is a fixed-length character-string of length n, each character 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 CARD_IMAGE,
      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(CARD_IMAGE);
```

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 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'.

Table 25 shows examples of digit and decimal-point characters.

Table 25. 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 characters

The picture characters Z and * 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 character. 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 9, T, I, R, or Y 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 * 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 9, T, I, R, or Y in a field.

Table 26 shows examples of zero suppression characters.

Table 26 (Page 1 of 2). 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	bb100
FIXED(5)	00100	ZZZZZ	bb100

Table 26 (Page 2 of 2). Examples of zero suppression characters

Source attributes	Source data (in constant form)	Picture specification	Character value
FIXED(5)	00000	ZZZZZ	bbbbb
FIXED(5,2)	123.45	ZZZ99	bb123
FIXED(5,2)	001.23	ZZZV99	bb123
FIXED(5)	12345	ZZZV99	undefined
FIXED(5,2)	000.08	ZZZVZZ	bbb08
FIXED(5,2)	000.00	ZZZVZZ	bbbbb
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 * appears to the right of the picture character V, all fractional digit positions in the specification, as well as all integer digit positions, must employ the Z or * 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 (B) 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.

, Inserts a comma into the associated position of the numeric character data when no zero suppression occurs. If zero suppression occurs, the comma is inserted only:

- When an unsuppressed digit appears to the left of the comma position
- When a V appears immediately to the left of the comma and the fractional part contains any significant digits
- When the comma is at the start of the picture specification
- When the comma is preceded only by characters not specifying digit positions

In all other cases where zero suppression occurs, the comma insertion character is treated as though it were a zero suppression character identical to the one immediately preceding it.

. Used the same way as the comma picture character, except that a point (.) is assigned to the associated position.

/ Used the same way as the comma picture character, except that a slash (/) is inserted in the associated position.

B Specifies that a blank character always be inserted into the associated position of the character value of the numeric character data.

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 various 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 numbers of nine digits 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. The character value of A, B, and C, respectively, is:

```
1,234.00
1.234,00
1 234,00
```

In the following example, when the assignment is executed, decimal point alignment 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;
```

Table 27 shows examples of insertion characters.

Table 27. 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	bbb03
FIXED(4,2)	00.03	ZZV.ZZ	bb.03
FIXED(4,2)	12.34	ZZV.ZZ	12.34
FIXED(4,2)	00.00	ZZV.ZZ	bbbbb
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	bbb12/34
FIXED(6)	000012	ZZ/ZZ/ZZ	bbbbbb12
FIXED(6)	000000	ZZ/ZZ/ZZ	bbbbbbbb
FIXED(6)	000000	**/**/**	********
FIXED(6)	000000	**B**B**	**b**b**
FIXED(6)	123456	99B99B99	12b34b56
FIXED(3)	123	9BB9BB9	1bb2bb3
FIXED(2)	12	9BB/9BB	1bb/2bb

Signs and currency characters

The picture characters S, +, and – specify signs in numeric character data. The picture character $ specifies a currency symbol in the character value of numeric character data. Only one type of sign character can appear in each field. These picture characters can be used in either a static or a drifting manner.

The 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.

The 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 has appeared 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.

$ Specifies the currency symbol. The static character must appear either to the left or right of all digit positions in a field of a specification. See details above for the drifting use of the character.

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.

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';
```

50 is held as '+050', zero as '+000' and -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.

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.

Table 28 shows examples of signs and currency symbol characters.

Table 28. 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+
FIXED(5,2)	001.23	++B+9V.99	bbb+1.23
FIXED(5,2)	001.23	---9V.99	bbb1.23
FIXED(5,2)	-001.23	SSS9V.99	bb-1.23

Credit, debit, overpunched, and zero replacement characters

The picture characters CR, DB, T, I, and R 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 Used the same way that CR is used except that the letters DB appear in the associated positions.

Any of the picture characters T, I, or R (known as overpunch characters) specifies that an EBCDIC character represents the corresponding digit and the sign of the data item.

This representation has arisen from the custom of indicating signs in numeric data held on punched cards, by overpunching a 12-punch (to represent +) or an 11-punch (to represent –) near the top of a card column containing a digit (usually the last column in a field). The resulting EBCDIC card-code is, in most cases, the same as that for an alphabetic character (as shown in Table 29). The 12-0 and 11-0 combinations are not characters in the PL/I set but are within the set of the 256 characters of the EBCDIC code.

Only one overpunched sign can appear in a specification for a fixed-point number. A floating-point specification can contain two—one in the mantissa field and one in the exponent field. The overpunch character can be specified for any digit position within a field.

The T, I, and R picture characters specify how the input characters are interpreted, as shown in Table 29 on page 314.

Table 29. Interpretation of the T, I, and R picture characters

T or I		T or R		Digit
Digit with +		Digit with –		
EBCDIC character	EBCDIC card code	EBCDIC character	EBCDIC card code	
{	12-0	}	11-0	0
A	12-1	J	11-1	1
B	12-2	K	11-2	2
C	12-3	L	11-3	3
D	12-4	M	11-4	4
E	12-5	N	11-5	5
F	12-6	O	11-6	6
G	12-7	P	11-7	7
H	12-8	Q	11-8	8
I	12-9	R	11-9	9

T, I, and R specify the following values:

T On input, T specifies that the EBCDIC characters { through I and the digits 0 through 9 represent positive values, and the EBCDIC characters } through R represent negative values.

On output, T specifies that the associated position contains one of the EBCDIC characters { through I if the input data represents positive values and one of the EBCDIC characters } through R if the input data represents negative values. The T can appear anywhere a '9' picture specification character occurs. For example:

```
DCL CREDIT PICTURE 'ZZV9T';
```

The character representation of CREDIT is 4 characters. +21.05 is held as '210E'. -0.07 is held as 'bb0P'.

I On input, I specifies that the EBCDIC characters { through I and the digits 0 through 9 represent positive values.

On output, I specifies that the associated position contains one of the EBCDIC characters { through I if the input data represents positive values; otherwise, it contains one of the digits 0 through 9.

R On input, R specifies that the EBCDIC characters } through R represent negative values and the digits 0 through 9 represent positive values.

On output, R specifies that the associated position contains one of the EBCDIC characters } through R if the input data represents negative values; otherwise, it contains one of the digits 0 through 9. For example:

```
DECLARE X FIXED DECIMAL(3);
GET EDIT (X) (P'R99');
```

sets X to 132 on finding '1327' in the next 3 positions of the input stream, but -132 on finding 'J32'.

The Y picture character specifies that zero is replaced by the blank character.

Y Specifies that a zero in the specified digit position is replaced unconditionally by the blank character.

Table 30 shows examples of credit, debit, overpunched, and zero replacement characters.

Table 30. Examples of credit, debit, overpunched, 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	$12.34bb
FIXED(4,2)	-12.34	$ZZV.99DB	$12.34DB
FIXED(4,2)	12.34	$ZZV.99DB	$12.34bb
FIXED(4)	1021	999I	102A
FIXED(4)	-1021	Z99R	102J
FIXED(4)	1021	99T9	10B1
FIXED(5)	00100	YYYYY	bb1bb
FIXED(5)	10203	9Y9Y9	1b2b3
FIXED(5,2)	000.04	YYYVY9	bbbb4

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).

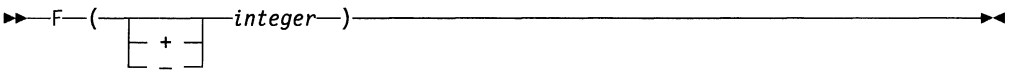
Table 31 shows examples of exponent characters.

Table 31. 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	+123.00Eb-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	+123.00Eb02

Scaling factor character

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.

Table 32 shows examples of the picture scaling factor character.

Table 32. 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 14. Condition handling

Chapter 14. Condition handling

When a PL/I program is executed, a number of conditions are detected if they are raised. These conditions can be errors, such as overflow or an input/output transmission error, or they can be conditions that are expected, such as the end of an input file or the end of a page when output is being printed. A condition is also raised when a SIGNAL statement for that condition is executed.

A condition is *enabled* when its raising will execute an action. An action specified to be executed when an enabled condition is raised is *established*. You use the enabling of conditions and the specifying of the action required when a condition is raised to control the handling of conditions. For descriptions of each of the conditions, see Chapter 15, "Conditions" on page 332.

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 16, "Built-in functions, subroutines, and pseudovariables" on page 360.

The implicit action for many conditions is raising 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 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 provides a fixed-point binary value of precision (15,0) whose value 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 15, "Conditions" on page 332.

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. For example:

```
(SIZE):  L1:  X=(I**N) / (M+L);
```

A condition in a prefix list indicates the corresponding condition is enabled within the scope of the prefix. A condition prefix can be attached to any statement except a DECLARE, DEFAULT, ENTRY, or % statement.

Some conditions are always enabled unless they are explicitly disabled by condition prefixes; others are always disabled unless they are explicitly enabled by condition prefixes; and still others are always enabled and cannot be disabled.

The conditions that are always enabled unless they are explicitly disabled by condition prefixes are:

CONVERSION
FIXEDOVERFLOW
OVERFLOW
UNDERFLOW
ZERODIVIDE

Each of the preceding conditions can be disabled by a condition prefix specifying the condition name preceded by NO without intervening blanks. Thus, one of the following in a condition prefix disables the respective condition:

NOCONVERSION
NOFIXEDOVERFLOW
NOOVERFLOW
NOUNDERFLOW
NOZERODIVIDE

Such a condition prefix renders the corresponding condition disabled throughout the scope of the prefix. The condition remains enabled outside this scope.

The conditions that are always disabled unless they are enabled by a condition prefix are:

SIZE
SUBSCRIPTRANGE
STRINGRANGE
STRINGSIZE

The appearance of one of these in a condition prefix renders the condition enabled throughout the scope of the prefix. The condition remains disabled outside this scope.

One of the following in a condition prefix disables the corresponding condition throughout the scope of that prefix:

NOSIZE
NOSUBSCRIPTRANGE
NOSTRINGRANGE
NOSTRINGSIZE

All other conditions are always enabled and cannot be disabled. These conditions are:

AREA	KEY
CONDITION	NAME
ENDFILE	RECORD
ENDPAGE	TRANSMIT
ERROR	UNDEFINEDFILE
FINISH	

Conditions that are detected by the compiler while compiling your program are diagnosed and do not raise the condition when the program is executed.

For example:

```
DCL A FIXED DEC(2);
    A = 999;
```

results in a message from the compiler whether SIZE is enabled or not.

Scope of the condition prefix

The scope of a condition prefix (the part of the program throughout 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 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.

The scope of a condition prefix applied to a DO or SELECT statement is limited to execution of the statement itself; it does not apply to execution of the entire group.

Condition prefixes attached to a compound statement do not apply to the statement or statements contained in the statement body of the compound statement.

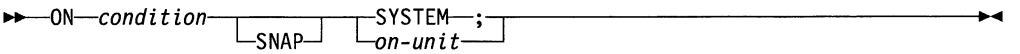
The enabling or disabling of a condition can be redefined within a block by attaching a prefix to statements within the block, including PROCEDURE and BEGIN statements (thus redefining the enabling or disabling of the condition within nested blocks). Such a redefinition applies only to the execution 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.

Established action

An implicit action exists for every condition, and if an enabled condition is raised, this implicit action is executed unless an ON-unit specified in an ON statement executed for that condition is in effect.

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 action. The syntax for the ON statement is:



condition

Any one of the conditions described in Chapter 15, "Conditions" on page 332.

SNAP Specifies that when the enabled condition is raised, a list is printed of all the blocks and ON-units active at the time the condition is raised. 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 the list of active blocks.

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 15, "Conditions" on page 332.

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 an unlabeled simple statement, it can be any simple statement except BEGIN, DECLARE, DEFAULT, DO, END, ENTRY, FORMAT, LEAVE, OTHERWISE, PROCEDURE, RETURN, SELECT, WHEN, or % statements. If an ON-unit is a single statement, it cannot refer to a remote format specification. If the ON-unit is an unlabeled 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.

Because the ON-unit itself requires a semicolon, no semicolon is shown for the ON-unit in the syntax.

An ON-unit is treated as a procedure (without parameters) 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 15, "Conditions" on page 332.

ON-units, except certain single-statement ON-units, are treated as separate program blocks by the compiler. They are separated from the ON statement and compiled with prologue and epilogue code. In order to save the overhead of executing prologue and epilogue code, certain single-statement ON-units are not compiled. Instead, the action required is carried out under the control of the error handling routine. The types of ON-units involved are:

- Null ON-units
- ON-units containing only SNAP, SNAP SYSTEM, or SYSTEM options
- ON-units containing only a GOTO statement

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 not the same as disabling, 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 system must still check for the raising of the condition).

Scope of the established action

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.

Note: Dynamic descendancy refers to the fact that ON-units are inherited from the calling procedure in all circumstances. Dynamic descendancy is often not known until run time.

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 descendent 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 reexecution of an overridden ON statement).

Dynamically-descendant ON-units

It is possible to raise a condition during execution of an ON-unit and enter a further ON-unit. An ON-unit entered due to a condition either raised or signalled in another ON-unit is a dynamically-descendant ON-unit. A normal return from a dynamically-descendant ON-unit reestablishes the environment of the ON-unit in which the condition was raised.

To prevent an ERROR condition raised in an ERROR ON-unit from executing the same ERROR ON-unit, thus raising the ERROR condition again and causing a loop, the following technique can be used:

```
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, since the ON-unit refers to file FC1. If the condition is raised in the second READ statement, however, the ON-unit is not entered, since 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.

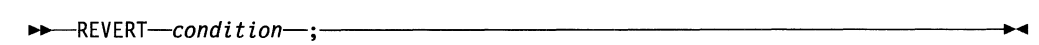
Example 4:

```
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 action specification of any ON statement for the condition that executed in that block; it then reestablishes the action specification that was in force at the time of activation of the block. It can affect only ON statements that are internal to the block in which the REVERT statement occurs and which have been executed in the same invocation of that block. The syntax for the REVERT statement is:



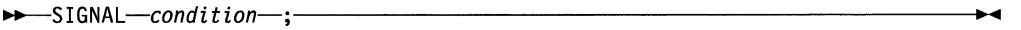
condition
Any one of the conditions described in Chapter 15, "Conditions" on page 332.

The execution of a REVERT statement has the effect described above only if (1) an ON statement, specifying the same condition and internal to the same invocation of the same block, was executed after the block was activated, and (2) the execution of no other similar REVERT statement has intervened. 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:

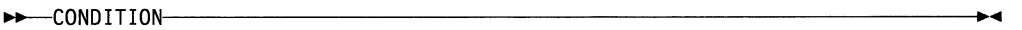


condition
Any one of the conditions described in Chapter 15, "Conditions" on page 332.

The CONDITION condition can be raised only as a result of its specification in a SIGNAL statement.

CONDITION attribute

The CONDITION attribute specifies that the name is a condition name. 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.

Multiple conditions

A *multiple condition* is the simultaneous raising of two or more conditions. A *multiple condition* can occur only for the conditions TRANSMIT and RECORD. The TRANSMIT condition is always processed first. The RECORD condition is ignored unless there is a normal return from the TRANSMIT ON-unit.

Event input/output conditions cannot be raised as part of the same multiple condition.

Multiple conditions are processed successively, until one of the following occurs, in which case no subsequent conditions are processed:

- The processing of a condition terminates the program, through either implicit action for the condition, normal return from an ON-unit, or abnormal termination in the ON-unit.
- Control is transferred out of an ON-unit by means of a GO TO statement, so that a normal return is not allowed.

Example of use of conditions

The routine shown in Figure 17 illustrates the use of the ON, SIGNAL, and REVERT statements, and condition prefixes.

The routine reads batches of records containing test readings. Each batch has a header record with a sample number, called SNO, of zero and a trailer record with SNO equal to 9999. If a CONVERSION condition is raised, one retry is made with the error character set to zero. Data fields are used to calculate subscripts; if a subscript is out of range, the sample is ignored. If there is more than one subscript error or more than one conversion error in a batch, that batch is ignored.

The numbers shown to the right of each line are sequence numbers, which are not part of the program itself.

The first statement executed within this routine is the ON ENDFILE statement in line 9. This specifies that the external procedure SUMMARY is to be called when an ENDFILE (PDATA) condition is raised. This action applies within DIST and within all other procedures called by DIST, unless they establish their own action for ENDFILE (PDATA).

```
/* SAMPLE ROUTINE EXHIBITING USE OF CONDITIONS */
    DIST: PROCEDURE;                                                    02
          DECLARE 1 SAMPLE EXTERNAL,                                    03
                    2 BATCH CHARACTER(6),                               04
                    2 SNO PICTURE '9999',                               05
                    2 READINGS CHARACTER(70),                           06
                  TABLE(15,15) EXTERNAL, (ONCHAR,ONCODE) BUILTIN;       07

   /* ESTABLISH INTERRUPT ACTIONS FOR ENDFILE & CONVERSION */           08
          ON ENDFILE (PDATA) CALL SUMMARY;                              09
          ON CONVERSION BEGIN;  CALL SKIPBCH;                           10
                                GO TO NEWBATCH;                         11
                        END;                                            12
          ON ERROR BEGIN;                                               13
             ON ERROR SYSTEM;                                           14
             DISPLAY  (BATCH||SNO||READINGS);                           15
             END;                                                       16

   /* MAIN LOOP TO PROCESS HEADER & TABLE */                            17
  HEADER: READ INTO (SAMPLE) FILE (PDATA);                              18
          /* CHECK ACTION LISTS INPUT DATA FOR DEBUG */                 19
          IF SNO ¬= 0 THEN SIGNAL CONVERSION;                           20
NEWBATCH: GET LIST (OMIN,OINT,AMIN,AINT) STRING (READINGS);             21
          TABLE = 0;                                                    22
          CALL INPUT;                                                   23
          CALL PROCESS;                                                 24
          GO TO HEADER;                                                 25
   /* ERROR RETURN FROM INPUT */                                        26
BADBATCH: SIGNAL CONVERSION;                                            27

/* READ RECORD BATCH WITH TEST READINGS AND UPDATE TABLE */
           INPUT: PROCEDURE;                                            29
           /* ESTABLISH CONDITION ACTIONS FOR CONVERSION & SUBRG */     30
                 ON CONVERSION BEGIN;                                   31
                               IF ONCODE = 624 & ONCHAR = ' '           32
                                  THEN DO; ONCHAR = '0';                33
                                           GO TO ERR1;                  34
                                       END;                             35
                                  ELSE GO TO BADBATCH;                  36
                               END;                                     37
                 ON SUBSCRIPTRANGE GO TO ERR2;                          38
           /* LOOP TO READ SAMPLE DATA AND ENTER IN TABLE */            39
             IN1: READ INTO (SAMPLE) FILE (PDATA);                      40
                  IF SNO = 9999 THEN RETURN;    /*TRAILER RECORD*/     41
             IN2: GET EDIT (R,OMEGA,ALPHA)(3 P'999')                    42
                      STRING (READINGS);                                43
(SUBSCRIPTRANGE): TABLE((OMEGA-OMIN)/OINT,(ALPHA-AMIN)/AINT) = R;       44
                  GO TO IN1;                                            45
       /* FIRST CONVERSION & SUBSCRIPTRANGE ERROR IN THIS BATCH */      46
           ERR2: ON SUBSCRIPTRANGE GO TO BADBATCH;/*FOR NEXT ERROR */   47
                 CALL ERRMESS(SAMPLE,02);                               48
                 GO TO IN1;                                             49
      ERR1: REVERT CONVERSION;      /*SWITCH FOR NEXT ERROR*/           50
            CALL ERRMESS(SAMPLE,01);                                    51
            GO TO IN2;                                                  52
            END INPUT;                                                  53
   END DIST;                                                            54
```

Figure 17. A program checkout routine

Throughout the procedure, any conditions except SIZE, SUBSCRIPTRANGE, STRINGRANGE, and STRINGSIZE are enabled by default, and, for all conditions except those mentioned explicitly in ON statements, the implicit action for the condition applies. This implicit action, in most cases, is to generate a message and then to raise the ERROR condition. The action specified for the ERROR condition in line 13 is to display the contents of the line being processed. When the ERROR action is completed, the FINISH condition is raised, and execution of the program is subsequently terminated.

The statement in line 10 specifies action to be executed whenever a CONVERSION condition is raised.

The main loop of the program starts with the statement labeled HEADER. This READ statement reads a record into the structure SAMPLE. If the record read is not a header, the SIGNAL CONVERSION statement causes execution of the BEGIN block, which in turn calls a procedure (not shown here) that reads on, ignoring records until it reaches a header. The begin block ends with a GO TO statement that terminates the ON-unit.

The GET statement labeled NEWBATCH uses the STRING option to get the different test numbers that have been read into the character string READINGS. Since the variables named in the data list are not explicitly declared, they are declared implicitly with the attributes FLOAT DECIMAL (6).

The array TABLE is initialized to zero before the procedure INPUT is called. This procedure inherits the ON-units already established in DIST, but it can override them.

The first statement of INPUT establishes a new action for the CONVERSION condition. Whenever a CONVERSION condition is raised, the ONCODE is tested to check that the raising of the condition is due to an invalid P-format input character and that the invalid character is a blank. If the invalid character is a blank, it is replaced by a zero, and control is transferred to ERR1.

ERR1 is internal to the procedure INPUT. The statement, REVERT CONVERSION, nullifies the ON CONVERSION statement executed in INPUT and restores the action specified for the CONVERSION condition in DIST (which causes the batch to be ignored).

After a routine is called to write an error message, control goes to IN2, which retries the conversion. If another CONVERSION condition is raised, the condition action is that specified in lines 10 and 11.

The second ON statement in INPUT establishes the action for a SUBSCRIPTRANGE condition. This condition must be explicitly enabled by a SUBSCRIPTRANGE prefix. If a SUBSCRIPTRANGE condition is raised, the ON-unit transfers to ERR2, which establishes a new ON-unit for SUBSCRIPTRANGE conditions, overriding the action specified in the ON statement in line 38. Any subsequent SUBSCRIPTRANGE conditions in this batch will, therefore, cause control to go to BADBATCH, which signals the CONVERSION condition as it existed in the procedure DIST. On leaving INPUT, the on-action reverts to that established in DIST, which in this case calls SKIPBCH to get to the next header.

After establishment of a new ON-unit, a message is printed, and a new sample record is read.

The statement labeled IN1 reads an 80-byte record image into the structure SAMPLE. A READ statement does not check input data for validity, so the CONVERSION condition cannot be raised.

The statement IN2 checks and edits the data in record positions 11 through 19 according to the picture format item. A nonnumeric character (including blank) in

these positions will raise a CONVERSION condition, with the results discussed above.

The next statement (line 44) has a SUBSCRIPTRANGE prefix. The data just read is used to calculate two subscripts. If either subscript falls outside the bounds declared for TABLE, a SUBSCRIPTRANGE condition is raised. If both fall outside the range, only one condition is raised. (The ON-unit, line 38, transfers control to ERR2 and the subscript calculation is not completed).

Chapter 15. Conditions

Chapter 15. Conditions

This chapter describes conditions in alphabetic order. In general, the following information is given for each condition:

- A discussion of the condition, including its syntax and the circumstances under which the condition can be raised. A condition can always be raised by a SIGNAL statement; this fact is not included in the descriptions.
- **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.
- **Implicit action**—the action taken when an enabled condition is raised, and no ON-unit is currently established for the 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.
- **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.

Classification of conditions

The conditions are classified as follows:

- *Computational conditions*—those associated with data handling, expression evaluation, and computation. The conditions are:

  CONVERSION SIZE
  FIXEDOVERFLOW UNDERFLOW
  OVERFLOW ZERODIVIDE

  If a computational condition (except UNDERFLOW) is raised and the condition is disabled, the program is in error.

- *Input/output conditions*—those conditions associated with input and output. They are:

  ENDFILE RECORD
  ENDPAGE TRANSMIT
  KEY UNDEFINEDFILE
  NAME

- *Program-checkout conditions*—those conditions that facilitate the debugging of a program. They are:

  STRINGSIZE STRINGRANGE
  SUBSCRIPTRANGE

  If SUBSCRIPTRANGE is raised and is disabled, the program is in error.

Because this checking involves a substantial overhead in both storage space and run time, it usually is used only in program testing—it is removed for production programs, because the above are normally disabled conditions.

- *Miscellaneous conditions*, which are:

  AREA ERROR
  FINISH CONDITION

Conditions

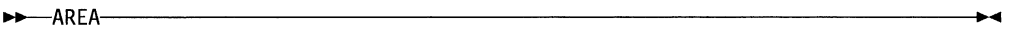
The following is a summary of all conditions in alphabetic sequence. The codes are shown for each condition. An explanation for each code is given under "Condition codes" on page 347.

AREA condition

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:



Result: In both cases the attempted allocation or assignment has no effect.

Implicit action: A message is printed and the ERROR condition is raised.

Status: AREA is always enabled.

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 reattempted.
- If the condition was raised by an allocation, the allocation is reattempted. 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 reattempted 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

CONDITION condition

The CONDITION condition is raised by a SIGNAL statement that specifies the appropriate name. The name specified in the SIGNAL statement determines which CONDITION condition is to be raised. The syntax for CONDITION is:

Abbreviation: COND

The CONDITION condition allows you to establish an ON-unit that will be executed whenever a SIGNAL statement is executed specifying CONDITION and that name.

As a debugging aid, this condition can be used to establish an ON-unit whose execution results in printing information that shows the current status of the program. The ON-unit can be executed from any point in the program through placement of a SIGNAL statement. Of course, normal rules of name scope apply; a condition name is external by default, but can be declared INTERNAL.

Following is an example of how the CONDITION condition might be included in a program:

```
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.

Status: CONDITION is always enabled.

Normal return: Execution continues with the statement following the SIGNAL statement.

Condition code: 500

CONVERSION condition

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 coded arithmetic, contains characters which are not the representation of an optionally signed arithmetic constant, or an expression to represent a complex constant.

  See "Loop handling" on page 335.
- 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, of course, that CONVERSION is not disabled). If the action specification is an ON-unit, the invalid character can be replaced within the unit by using the ONSOURCE or ONCHAR pseudovariables.

If the CONVERSION condition is raised and it is disabled, the program is in error.

If the CONVERSION condition is raised under graphic conditions (that is, GRAPHIC built-in), ONCHAR and ONSOURCE do not contain valid results. If the program attempts a normal return under these conditions, the ERROR condition is raised.

Result: When CONVERSION is raised, the contents of the entire result field are undefined.

Loop handling: An infinite loop can occur from either of the following two situations:

1. If you are converting from a character string to a numeric, and you use a character string containing an *E* or an *F*, the system can interpret the *E* as part of a legitimate number in exponential notation, or the *F* as a scaling factor. The combination of the *E* or *F* with other nonnumeric characters can result in an infinite loop in the error handler.
2. If you are converting from a character string to a numeric, and the character string ends with the letter *B*, the CONVERSION routine assumes that the field is fixed binary. This can also result in an infinite loop.

It might be helpful to use ONSOURCE instead of ONCHAR in the conversion ON-unit. Set ONSOURCE to *0* when conversion is initially raised, thus avoiding the loop (see Chapter 16, "Built-in functions, subroutines, and pseudovariables" on page 360).

Implicit action: A message is printed and the ERROR condition is raised.

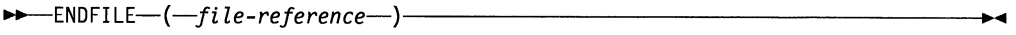
Status: CONVERSION is enabled throughout the program, except within the scope of a condition prefix specifying NOCONVERSION.

Normal return: If the ONSOURCE or ONCHAR pseudovariable is used, the program retries the conversion on return from the ON-unit. *If the error is not corrected, the program will loop.* If these pseudovariables are not used, the ERROR condition is raised.

Condition codes: 600-639

ENDFILE condition

The ENDFILE input/output condition can be raised during a GET, READ, or WAIT 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.

If the file is not closed after ENDFILE is raised, any subsequent GET or READ statement for that file immediately raises the ENDFILE condition again.

The ENDFILE condition for a data transmission statement using the EVENT option is raised when the WAIT statement for that event is encountered.

Implicit action: A message is printed and the ERROR condition is raised.

Status: The ENDFILE condition is always enabled.

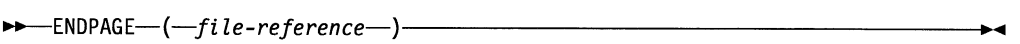
Normal return: Execution continues with the statement immediately following the GET or READ statement that raised the ENDFILE (or, if ENDFILE was raised by a WAIT statement, control passes back to the WAIT statement).

If a file is closed in an ON-unit for this condition, the results of normal return are undefined. Exit from such an ON-unit must be by means of a GO TO statement.

Condition code: 70

ENDPAGE condition

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. The syntax for ENDPAGE is:



ENDPAGE is raised only once per page, except when it is raised by the SIGNAL statement.

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.

Implicit action: A new page is started. If the condition is signaled, execution is unaffected and continues with the statement following the SIGNAL statement.

Status: ENDPAGE is always enabled.

Normal return: Execution of the PUT statement continues in the manner described above.

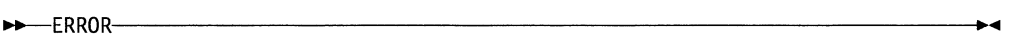
Condition code: 90

ERROR condition

The ERROR condition is raised under the following circumstances:

- As a result of the implicit action for a condition for which that action is to *print an error message and raise the ERROR condition.*
- As a result of an error (for which there is no other condition) during program execution.
- As a result of an abend.
- As a result of a SIGNAL ERROR statement.

The syntax for ERROR is:



Implicit action:

- If a message has not already been printed as a result of an implicit action for some other condition, then a message is printed describing the error. In all cases, the FINISH condition is raised and the program terminates.

Status: ERROR is always enabled.

Normal return: The implicit action is taken.

Condition codes: Code 3, code 9, and all codes 1000 and above are ERROR conditions.

FINISH condition

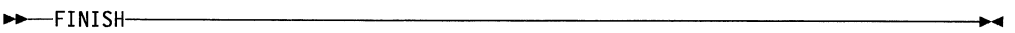
The FINISH condition is raised during execution of a statement that would terminate the PL/I program, that is, by a STOP or EXIT statement in any procedure, or a RETURN or END statement in the MAIN procedure of the program. The condition is also raised by SIGNAL FINISH, and as part of the implicit action for the ERROR condition. An abnormal return from the ON-unit avoids program termination and allows the program to continue.

When a program is made up of PL/I and non-PL/I 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.

For information on communication between PL/I and non-PL/I procedures, see the *LE/VSE Programming Guide.*

The syntax for FINISH is:



Implicit action: No action is taken and processing continues from the point where the condition was raised.

Status: FINISH is always enabled.

Normal return: Execution of the statement is resumed.

Condition code: 4

FIXEDOVERFLOW condition

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.

Result: The result of the invalid fixed-point operation is undefined.

Implicit action: A message is printed and the ERROR condition is raised.

Status: FIXEDOVERFLOW is enabled throughout the program, except within the scope of a condition prefix that specifies NOFIXEDOVERFLOW.

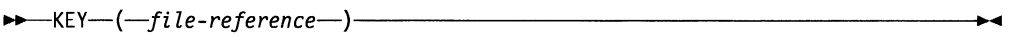
Normal return: Control returns to the point immediately following the point at which the condition was raised.

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.

KEY condition

The KEY input/output condition can be raised only during operations on keyed records. It is raised in the cases mentioned in the list of condition codes, below. The syntax for KEY is:



When a LOCATE statement is used for a VSAM key-sequenced data set, the KEY condition for this LOCATE statement is not raised until transmission of the record is attempted; that is, at the next WRITE or LOCATE statement for the file, or when the file is closed.

The KEY condition for a data transmission statement using the EVENT option is raised when the WAIT statement for that event is encountered.

When a LOCATE statement is used for a REGIONAL(3) data set with V-format or U-format records, and there is not enough room in the specified region, the KEY condition is not raised until transmission of the record is attempted. Neither the record for which the condition is raised nor the current record is transmitted.

Implicit action: A message is printed and the ERROR condition is raised.

Status: KEY is always enabled.

Normal return: Control passes to the statement immediately following the statement that raised KEY (or, if KEY was raised by a WAIT statement, control passes back to the WAIT statement).

If a file is closed in an ON-unit for this condition, the results of normal return are undefined. Exit from such an ON-unit should be by means of a GO TO statement.

Condition codes: 50-58

NAME condition

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 "Data-directed element assignments" on page 280.

- 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.
  - The name is iSUB-defined.
  - DBCS contains a byte outside the valid range of 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.

Status: NAME is always enabled.

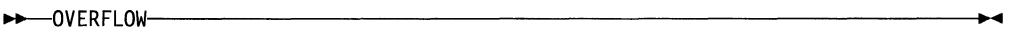
Normal return: The execution of the GET statement continues with the next name in the stream.

Condition code: 10

OVERFLOW condition

The OVERFLOW computational condition is raised when the magnitude of a floating-point number exceeds the maximum allowed. The magnitude of a floating-point number or intermediate result must not be greater than 10^{75} or 2^{252}.

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.

Result: The value of such an invalid floating-point number is undefined.

Implicit action: A message is printed and the ERROR condition is raised.

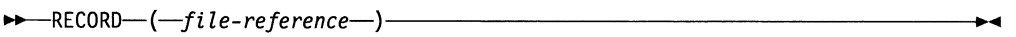
Status: OVERFLOW is enabled throughout the program, except within the scope of a condition prefix specifying NOOVERFLOW.

Normal return: Control returns to the point immediately following the point at which the condition was raised.

Condition code: 300

RECORD condition

The RECORD input/output condition can be raised only during a READ, WRITE, LOCATE, or REWRITE operation. It is raised in the cases mentioned under "Condition codes" below. The syntax for RECORD is:



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.

The RECORD condition for a data transmission statement using the EVENT option is raised when the WAIT statement for that event is encountered.

The RECORD condition is not raised for undefined-length records read from:

- A CONSECUTIVE data set through a SEQUENTIAL UNBUFFERED file
- A REGIONAL(3) data set through a DIRECT file

Implicit action: A message is printed and the ERROR condition is raised.

Status: RECORD is always enabled.

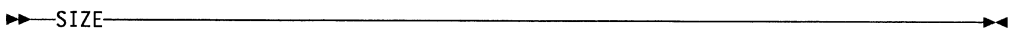
Normal return: Execution continues with the statement immediately following the one for which RECORD was raised (or if RECORD was raised by a WAIT statement, control returns to the WAIT statement).

If a file is closed in an ON-unit for this condition, the results of normal return are undefined. Exit from such an ON-unit should be by means of a GO TO statement.

Condition codes: 20-24

SIZE condition

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:



The SIZE condition differs from the FIXEDOVERFLOW condition in that, whereas FIXEDOVERFLOW is raised when the size of a calculated fixed-point value exceeds the maximum allowed by the implementation, 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. SIZE can be raised on assignment of a value regardless of whether or not FIXEDOVERFLOW was raised in the calculation of that value.

The declared size is not necessarily the actual precision with which the item is held in storage; however, the limit for SIZE is the declared or default size, not the actual size in storage. 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 this checking involves a substantial overhead in both storage space and run time, it usually is used only in program testing. You should remove it for production programs.

If the SIZE condition is raised and it is disabled, the program is in error.

Result: The result of the assignment is undefined.

Implicit action: A message is printed and the ERROR condition is raised.

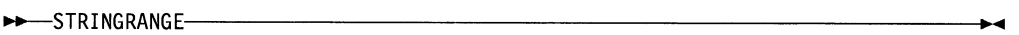
Status: SIZE is disabled within the scope of a NOSIZE condition prefix and elsewhere throughout the program, except within the scope of a condition prefix specifying SIZE.

Normal return: Control returns to the point immediately following the point at which the condition was raised.

Condition codes: 340 341

STRINGRANGE condition

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 such reference. The syntax for STRINGRANGE is:



Abbreviation: STRG

Implicit action: A message is printed and processing continues as described for normal return.

Status: STRINGRANGE is disabled by default and within the scope of a NOSTRINGRANGE condition prefix. It is enabled only within the scope of a STRINGRANGE condition prefix.

Normal return: Execution continues with a revised SUBSTR reference whose 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

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

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.

Implicit action: A message is printed and processing continues. However, if error messages and program output are using the same output stream, the output is unpredictable because no synchronization between them is provided.

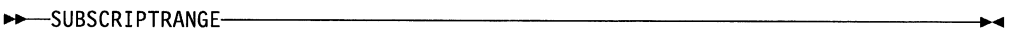
Status: STRINGSIZE is disabled by default and within the scope of a NOSTRINGSIZE condition prefix. It is enabled only within the range of a STRINGSIZE condition prefix.

Normal return: Execution continues from the point at which the condition was raised.

Condition codes: 150 151

SUBSCRIPTRANGE condition

The SUBSCRIPTRANGE program-checkout condition is raised whenever a subscript is evaluated and found to lie outside its specified bounds. The condition is also raised when an iSUB subscript is outside the range given in the declaration of the iSUB defined array. The order of raising SUBSCRIPTRANGE relative to evaluation of other subscripts is undefined. The syntax for SUBSCRIPTRANGE is:



Abbreviation: SUBRG

Result: When SUBSCRIPTRANGE has been raised, the value of the invalid subscript is undefined, and, hence, the reference is also undefined.

Implicit action: A message is printed and the ERROR condition is raised.

Status: SUBSCRIPTRANGE is disabled by default and within the scope of a NOSUBSCRIPTRANGE condition prefix. It is enabled only within the scope of a SUBSCRIPTRANGE condition prefix.

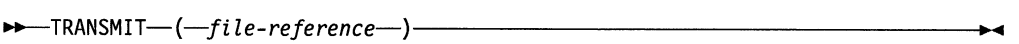
Normal return: Normal return from a SUBSCRIPTRANGE ON-unit raises the ERROR condition.

Condition codes: 520 521

TRANSMIT condition

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) and, therefore, signifies that any data transmitted is potentially incorrect.

Uncorrectable transmission error means an input/output error that could not be corrected during this 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.

The TRANSMIT condition for a data transmission statement using the EVENT option is raised when the WAIT statement for that event is encountered in the same process.

A CLOSE statement for an OUTPUT file can raise the TRANSMIT condition because the buffers might need to be transmitted before closing. In this event, a subsequent CLOSE statement in the ON-unit will be ignored. If control returns from the ON-unit by means of a GO TO statement, any remaining files in the original CLOSE statement will not be processed.

Implicit action: A message is printed and the ERROR condition is raised.

Status: TRANSMIT is always enabled.

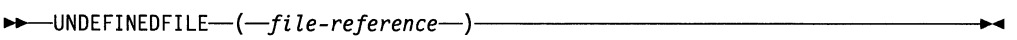
Normal return: Processing continues as though no error had occurred, allowing another condition (for example, RECORD) to be raised by the statement or data item that raised the TRANSMIT condition. (If TRANSMIT is raised by a WAIT statement, control returns to the WAIT statement).

If a file is closed in an ON-unit for this condition, the results of normal return are undefined. Exit from such an ON-unit should be by means of a GO TO statement.

Condition codes: 40-46

UNDEFINEDFILE condition

The UNDEFINEDFILE input/output condition is raised whenever a nonzero return code is received from the OPEN SVC. 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 files specified. 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 without the EVENT option, 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.

If UNDEFINEDFILE is raised by an implicit file opening in a data transmission statement having an EVENT option, the event variable retains its previous value and remains inactive. On normal return from the ON-unit, the event variable is initialized, that is, it is made active and its completion value is set to '0'B (provided the file has been opened in the ON-unit). Processing then continues with the remainder of the statement. However, if the file has not been opened in the ON-unit, the event variable remains uninitialized, the statement cannot be continued, and the ERROR condition is raised.

UNDEFINEDFILE is raised when the TOTAL option of the ENVIRONMENT attribute is specified and either attributes have been added on an OPEN statement or attributes implied by a data transmission statement conflict with default attributes.

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 REGIONAL(2) or REGIONAL(3) data sets
- Specifying a KEYLENGTH for a VSAM KSDS that conflicts with the actual data set value
- Specifying a V-format logical record length of less than 18 bytes for STREAM data sets
- Specifying, for FB-format records, a block size that is not an integral multiple of the record size
- Specifying, for VB-format records, a logical record length that is not at least 4 bytes smaller than the specified block size

Implicit action: A message is printed and the ERROR condition is raised.

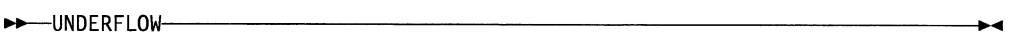
Status: UNDEFINEDFILE is always enabled.

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. (see UNDEFINEDFILE description on page 345, for action in the case of an implicit opening).

Condition codes: 80-89 91-95

UNDERFLOW condition

The UNDERFLOW computational condition is raised when the magnitude of a floating-point number is smaller than the minimum allowed. The magnitude of a nonzero floating-point value cannot be less than 10^{-78} or 2^{-260}. The syntax for UNDERFLOW is:



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.

Result: The invalid floating-point value is set to 0.

Implicit action: A message is printed, and execution continues from the point at which the condition was raised.

Status: UNDERFLOW is enabled throughout the program, except within the scope of a condition prefix specifying NOUNDERFLOW.

Normal return: Control returns to the point immediately following the point at which the condition was raised.

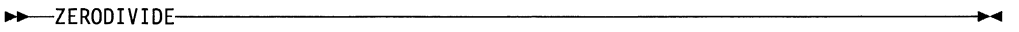
Condition code: 330

ZERODIVIDE condition

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. The compiler can also raise this condition, instead of FIXEDOVERFLOW, when:

- The results of a conversion from decimal to binary exceeds the maximum length allowed by the implementation.
- A fixed, floating-point, or decimal divide exception is detected by the hardware, as, for example, when using the DIVIDE built-in function and the quotient exceeds the size specified for the result.

The syntax for ZERODIVIDE is:



Abbreviation: ZDIV

If the ZERODIVIDE condition is raised and it is disabled, the program is in error.

Result: The result of a division by zero is undefined.

Implicit action: A message is printed and the ERROR condition is raised.

Status: ZERODIVIDE is enabled throughout the program, except within the scope of a condition prefix specifying NOZERODIVIDE.

Normal return: Control returns to the point immediately following the point at which the condition was raised.

Condition code: 320

Condition codes

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, STOP, or EXIT statement executed.

9 SIGNAL ERROR statement executed.

10 SIGNAL NAME statement executed or NAME condition occurred.

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 VSAM 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 (VSAM).

44 Uncorrectable transmission error on input from index set (VSAM).

45 Uncorrectable transmission error on output to sequence set (VSAM).

46 Uncorrectable transmission error on input from sequence set (VSAM).

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 REGIONAL or key-sequenced VSAM 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 a key-sequenced VSAM data set.

56 Attempt to access a record using a key that is outside the data set limits.

58 Key of record to be added lies outside the range(s) specified for the data set.

70 SIGNAL ENDFILE statement executed or ENDFILE condition occurred.

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 VSAM data set has not been loaded.

83 After merging ENVIRONMENT options with DLBL statement and data set label, data set specification is incomplete; for example, block size or record format has not been specified.

84 No DLBL 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 DLBL 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 DTF FUNCTION specified.

88 After merging ENVIRONMENT options with DLBL 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 or ENDPAGE condition occurred.

91 ENVIRONMENT option invalid for file accessing VSAM data set.

92 Error detected by VSAM while opening a VSAM data set; or during opening of a VSAM 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 a VSAM data set is empty.

150 SIGNAL STRINGSIZE statement executed or STRINGSIZE condition occurred.

151 Truncation occurred during assignment of a mixed character string.

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.

500 SIGNAL CONDITION (name) statement executed.

510 SIGNAL CHECK statement executed.

520 SIGNAL SUBSCRIPTRANGE statement executed, or subscript has been evaluated and found to lie outside its specified bounds.

521 Subscript of iSUB-defined variable lies outside bounds of corresponding dimension of base variable.

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.

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.

1011 Data management detected an input/output error but is unable to provide any information about its cause.

1012 A READ SET or READ INTO statement not preceded by a REWRITE.

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 maximum allowed.

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 task.

1020 Further input/output attempted before WAIT statement executed to ensure completion of previous READ.

1022 Unable to extend VSAM data set.

1024 Incorrect sequence of I/O operations on device-associated file.

1025 Insufficient virtual storage available for VSAM to complete request.

1026 No position established in VSAM data set.

1027 Record or VSAM control interval already held in exclusive control.

1028 Requested record lies on nonmounted volume.

1029 Attempt to reposition in VSAM data set failed.

1030 An error occurred during index upgrade on a VSAM data set.

1031 Invalid sequential write attempted on VSAM data set.

1040 A data set open for output used all available space.

1500 Computational error; short floating point argument of SQRT built-in function is negative.

1501 Computational error; long floating point argument of SQRT built-in function is < 0.

1502 Computational error; extended floating point argument of SQRT built-in function is negative.

1503 Computational error in LOG, LOG2, or LOG10 built-in function; extended floating point argument is <= 0.

1504 Computational error in LOG, LOG2, or LOG10 built-in function; short floating point argument is <= 0.

1505 Computational error in LOG, LOG2 or LOG10 built-in function; long floating point argument is <= 0.

1506 Computational error in SIN, COS, SIND, or COSD built-in function; absolute value of short floating point argument exceeds (2**18)*pi (SIN and COS) or (2**18)*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**50)*pi (SIN and COS) or (2**50)*180 (SIND and COSD).

1508 Computational error; absolute value of short floating point argument of TAN or TAND built-in function exceeds, respectively, (2**18)*pi or (2**18)*180.

1509 Computational error; absolute value of long floating point argument of TAN or TAND built-in function exceeds, respectively, (2**50)*pi or (2**50)*180.

1510 Computational error; short floating point arguments of ATAN or ATAND built-in function both zero.

1511 Computational error; long floating point arguments of ATAN or ATAND built-in function both zero.

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; absolute value of extended floating point argument exceeds (2**106)*pi (SIN and COS) or (2**106)*180 (SIND and COSD).

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 zero.

1522 Computational error; absolute value of extended floating point argument of TAN or TAND built-in function >= (2**106)*pi or (2**106)*180, respectively.

1550 Computational error; during exponentiation, real short floating-point base is zero and integer value exponent is not positive.

1551 Computational error; during exponentiation, real long floating-point base is zero and integer value 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 value exponent is not positive.

1555 Computational error; during exponentiation, complex long floating point base is zero and integer value 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 value 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 value 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.

2002 WAIT statement cannot be executed because of restricted system facility.

2050 A WAIT statement would cause a permanent wait.

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.

3001 Value of W field in F-format specification too small.

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 In an ON-unit entered as a result of the CONVERSION condition being raised by an invalid character in the string being converted, the character has not been corrected by use of the ONSOURCE or ONCHAR pseudovariables.

3800 Length of data aggregate exceeds system limit of 2**24 bytes.

3801 Array structure element not mapped.

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 task.

3909 Attempt to create a subtask (using CALL statement) when insufficient main storage available

3910 Attempt to attach a task (using CALL statement) when number of active tasks 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.

3920 An out-of-storage abend occurred.

4001 Attempt to assign data to an unallocated CONTROLLED variable during GET DATA.

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.

9050 Program terminated by an abend.

9200 Program check in SORT/MERGE program.

9250 Procedure to be fetched cannot be found.

9251 Permanent transmission error when fetching a procedure.

9253 Debugging tool unavailable.

9255 Attempt to release phase containing non-PL/I high-level language programs.

9258 Attempt to fetch a PL/I program not compiled by PL/I VSE.

9259 Module cannot be fetched because the entry point is a PL/I main.

9300 The CHECKPOINT/RESTART facility is not supported in a dynamic partition.

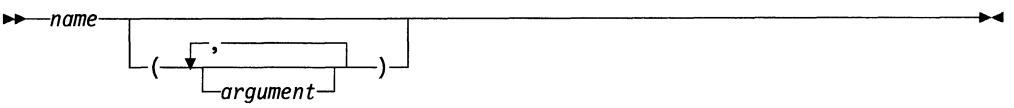
9301 The CHECKPOINT/RESTART facility is not supported in a partition that extends above the 16-megabyte line.

9999 A failure occurred during an invocation of an LE/VSE service.

Chapter 16. Built-in functions, subroutines, and pseudovariables

Chapter 16. Built-in functions, subroutines, and pseudovariables

The syntax of a built-in function or pseudovariable reference is:



The built-in functions, subroutines, and pseudovariables are listed in alphabetic order later in this chapter. In general, each description has the following:

- 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

Arguments, which can be expressions, are evaluated and converted to a data type suitable for the built-in function or pseudovariable according to the rules for data conversion.

The abbreviations for built-in functions and pseudovariables have separate declarations (explicit or contextual) and name scopes. In the following example:

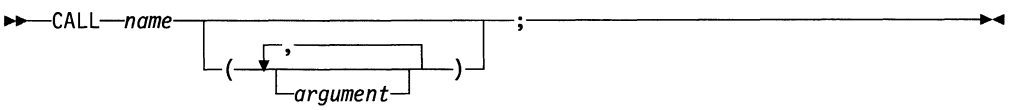
```
DCL (STG, STORAGE) BUILTIN;
```

is not a multiple declaration, and

```
DCL BINARY FILE;
X = BIN (VAR, 6,3);
```

is valid.

The syntax of a built-in subroutine is:



In general, each description has the following:

- A heading showing the syntax of the reference. The leading keyword CALL and trailing semicolon are omitted.
- Details of the arguments can be found in the *PL/I VSE Programming Guide.*
- Any other qualifications on the use of the subroutine.

Arguments, which can be expressions, are evaluated and converted to a data type suitable for the built-in subroutine according to the rules for data conversion. (This does not apply to PLITDLI.)

For a detailed description of the subroutines, see the *PL/I VSE Programming Guide.*

Classification of built-in functions

To aid in their description, built-in functions are listed in classes below. The first four classes are *computational built-in functions*.

String-handling
Arithmetic
Mathematical
Array-handling

Condition-handling
Storage control
Input/Output
Miscellaneous

String-handling built-in functions

The string-handling built-in functions simplify the processing of bit, character, and DBCS strings. They are:

BIT	HIGH	MPSTR	TRANSLATE
BOOL	INDEX	REPEAT	UNSPEC
CHAR	LENGTH	STRING	VERIFY
GRAPHIC	LOW	SUBSTR	

Note: The functions REPEAT, TRANSLATE, and VERIFY do not support GRAPHIC data.

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.

Arithmetic built-in functions

The arithmetic built-in functions allow you to:

1. Control conversion of base, scale, mode, and precision both directly and during basic arithmetic operations.
2. Determine properties of arithmetic values. For example, the SIGN function indicates the sign of an arithmetic value.

They are:

ABS	DECIMAL	IMAG	PRECISION
ADD	DIVIDE	MAX	REAL
BINARY	FIXED	MIN	ROUND
CEIL	FLOAT	MOD	SIGN
COMPLEX	FLOOR	MULTIPLY	TRUNC
CONJG			

Some of these 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 under "Data conversion in arithmetic operations" on page 69. When a data attribute of the result cannot agree with that of the argument (for example, the FLOOR built-in function), the rules are given in the function description.

Mathematical built-in functions

The mathematical built-in functions provide mathematical operations. They are:

ACOS	COSD	LOG	SINH
ASIN	COSH	LOG2	SQRT
ATAN	ERF	LOG10	TAN
ATAND	ERFC	SIN	TAND
ATANH	EXP	SIND	TANH
COS			

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 later in this chapter. Math routines with different accuracies are available. For additional information about these routines, see the *PL/I VSE Programming Guide.*

Array-handling built-in functions

The array-handling built-in functions operate on array arguments and return an element value. They are:

ALL	LBOUND
ANY	POLY
DIM	PROD
HBOUND	SUM

Any conversion of arguments required for these functions is noted in the function description.

Condition-handling built-in functions

The condition-handling built-in functions allow you to investigate the cause of enabled conditions. They are:

DATAFIELD	ONFILE
ONCHAR	ONKEY
ONCODE	ONLOC
ONCOUNT	ONSOURCE

Use of these functions is *in context* when within the scope of an ON-unit entered for the condition specific to the built-in function, or within an ON-unit for the ERROR or FINISH condition when raised as an implicit action. All other uses are *out of context.*

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. They are:

ADDR	ENTRYADDR	POINTERADD
ALLOCATION	NULL	POINTERVALUE
BINARYVALUE	OFFSET	STORAGE
CURRENTSTORAGE	POINTER	SYSNULL
EMPTY		

Input/output built-in functions

The input/output built-in functions allow you to determine the current state of a file. They are:

```
COUNT
LINENO
SAMEKEY
```

Miscellaneous built-in functions

The built-in functions that do not fit into any of the foregoing classes are:

```
DATE
DATETIME
PLIRETV
TIME
```

Built-in subroutines

The PL/I built-in subroutines are the following:

```
PLICANC        PLIREST        PLISRTB
PLICKPT        PLIRETC        PLISRTC
PLIDUMP        PLISRTA        PLISRTD
```

Note: PLITDLI cannot be declared with the BUILTIN attribute but is treated as a special subroutine.

Pseudovariables

Pseudovariables represent receiving fields. Except when noted in the description, the pseudovariable:

- Can appear on the left of the assignment symbol in an assignment or a do-specification
- Can appear in a data list of a GET statement or in the STRING option of a PUT statement
- Can appear as the string name in a KEYTO or REPLY option

Pseudovariables cannot be nested. For example, the following is invalid:

```
UNSPEC(SUBSTR(A,1,2)) = '00'B;
```

The pseudovariables are:

```
COMPLETION     ONCHAR         STRING
COMPLEX        ONSOURCE       SUBSTR
ENTRYADDR      REAL           UNSPEC
IMAG           STATUS
```

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, except for the POLY built-in function). ADDR, ALLOCATION, CURRENTSTORAGE, STORAGE, 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 arguments are ADDR, ALLOCATION, CURRENTSTORAGE, STORAGE, and STRING.

Null argument lists

Some built-in functions and pseudovariables 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 cannot be recognized by the compiler as a built-in function or pseudovariable name.

The built-in functions or pseudovariables that have no arguments or have a single optional argument are:

DATAFIELD	ONCODE	PLIRETV
DATE	ONCOUNT	STATUS
DATETIME	ONFILE	SYSNULL
EMPTY	ONKEY	TIME
NULL	ONLOC	
ONCHAR	ONSOURCE	

Descriptions of built-in functions, subroutines, and pseudovariables

ABS — Arithmetic

ABS returns the positive value of x, if x is real. If x is complex, ABS returns the positive square root of the sum of the squares of the real and imaginary parts. The syntax for ABS is:

x Expression.

The mode of the result is REAL. The result has the base, scale, and precision of x, except when x is fixed-point and complex with precision (p,q). The precision of the result is then given by:

```
(MIN(N,p+1),q)
```

where N is the maximum number of digits allowed.

ACOS — Mathematical

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.

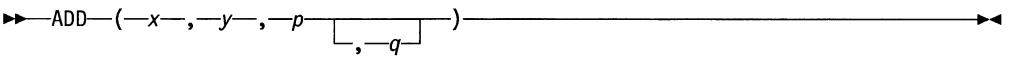
The result is in the range:

```
0<=ACOS(x)<=pi
```

and has the base and precision of x.

ADD — Arithmetic

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.

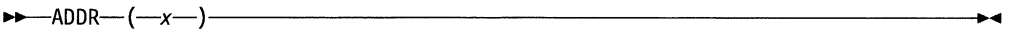
p Integer specifying the number of digits to be maintained throughout the operation; it must not exceed the implementation limit for the result base and scale.

q Optionally-signed integer 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 simply by prefixing the operand to be subtracted with a minus sign.

ADDR — Storage control

ADDR returns the pointer value that identifies the generation of x. The syntax for ADDR is:



x Reference 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 whose first base element is an unaligned fixed-length bit string (except where it is also the first element of the containing major structure)

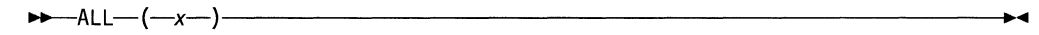
- A major structure that has the DEFINED attribute or is a parameter, and that has an unaligned fixed-length bit string as its first element
- A reference which is not to connected storage

If x is a reference to:

- An aggregate parameter, it must have the CONNECTED attribute or the CONTROLLED 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 qualification
- A varying string, the returned value identifies the 2-byte prefix
- An area, the returned value identifies the control information
- 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 — Array-handling

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:

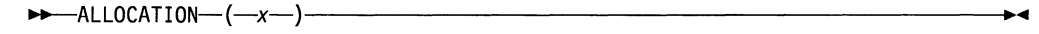


x Array expression.

If x is not a bit-string array, it is converted to bit string. If x is a reference to a defined variable, x must not be iSUB-defined.

ALLOCATION — Storage control

ALLOCATION returns a FIXED BINARY (31,0) value specifying the number of generations that can be accessed for x. The syntax for ALLOCATION is:



Abbreviation: ALLOCN

x Level-one unsubscripted controlled variable.

ANY — Array-handling

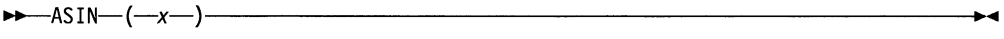
ANY returns a bit string in which each bit is 1 if the corresponding bit in any element of x exists and is 1. The length of the result is equal to that of the longest element. The syntax for ANY is:

x Array expression.

If x is not a bit-string array, it is converted to bit string. If x is a reference to a defined variable, x must not be iSUB-defined.

ASIN — Mathematical

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.

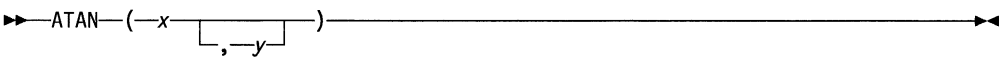
The result is in the range:

```
-pi/2<=ASIN(x)<=pi/2
```

and has the base and precision of x.

ATAN — Mathematical

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 and is real, the result is real, has the base and precision of x, and is in the range:

```
-pi/2<ATAN(x)<pi/2
```

If x alone is specified and is complex, it must not be +1I or -1I. The result is complex, has the base and precision of x, and a value given by:

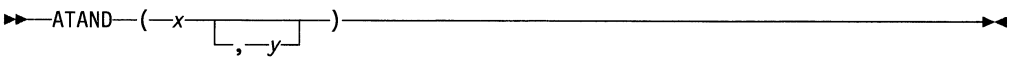
```
-1I*ATANH(1I*x)
```

If x and y are specified, each must be real. It is an error if x and y are both zero. The result for all other values of x and y is real, and 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
pi+ATAN(x/y)  for y<0 and x>=0
-pi+ATAN(x/y) for y<0 and x<0
```

ATAND — Mathematical

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<ATAND(x)<90
```

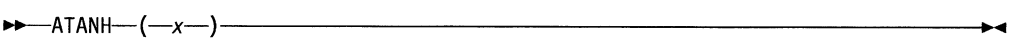
If x and y are specified, each must be real. The value of the result is given by:

```
(180/pi)*ATAN(x,y)
```

See the discussion of ATAN for argument requirements and attributes of the result.

ATANH — Mathematical

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:



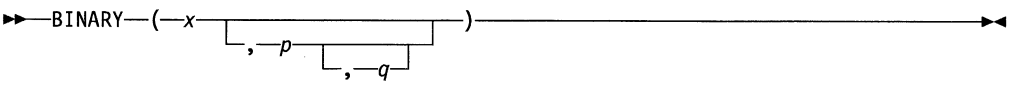
x Expression. If x is real, then ABS(x)<1. If x is complex, it must not be equal to +1 or -1.

The result has a value given by:

```
LOG((1+x)/(1-x))/2
```

BINARY — Arithmetic

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.

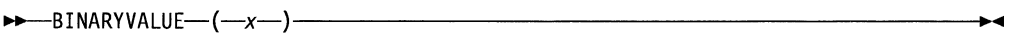
p Integer specifying the number of digits to be maintained throughout the operation; it must not exceed the implementation limit.

q Optionally-signed integer 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 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 — Storage control

BINARYVALUE returns a REAL FIXED BIN(31,0) value that is the converted value of its pointer expression, x. The syntax for BINARYVALUE is:

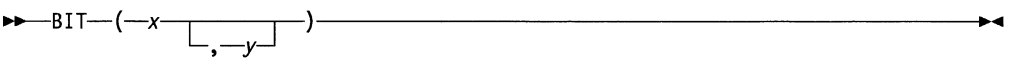


Abbreviation: BINVALUE

x Expression.

BIT — String-handling

BIT returns the bit value of x, with a length specified by y. The syntax for BIT is:

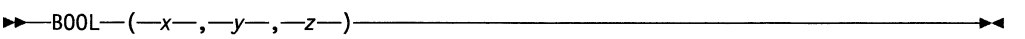


x Expression.

y Expression. If necessary, y is converted to a FIXED BINARY (15,0) 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 — String-handling

BOOL returns a bit string that is the result of a Boolean operation, specified by 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 — Arithmetic

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.

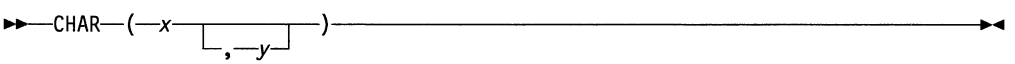
The result has the mode, base, scale, and precision of x, except when x is fixed-point with precision (p,q). The precision of the result is then given by:

```
(MIN(N,MAX(p-q+1,1)),0)
```

where N is the maximum number of digits allowed.

CHAR — String-handling

CHAR returns the character value of x, with a length specified by y. CHAR also supports conversion from graphic to character type. The syntax for CHAR is:



x Expression.

When x is non-GRAPHIC, CHAR returns x converted to character.

When x is GRAPHIC, CHAR returns x converted to mixed character, with the GRAPHIC data enclosed in shift-out/shift-in codes.

The values of x are not checked.

y Expression. If necessary, y is converted to a FIXED BINARY (15,0) 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 = 2, the result is a character string of 2 blanks.

If y = 3, the result is a character string of 3 blanks.

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, allowing space for and appending a shift-in code, and 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:

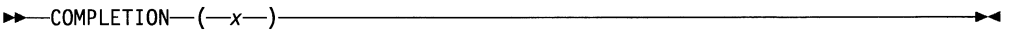
Conversion from graphic to character, where "y" is too short:

```
DCL X GRAPHIC(6);
DCL A CHAR (11);
A = CHAR(X);
```

For X with values:	Intermediate Result:	A is assigned:
.A.B.C.D.E.F	<.A.B.C.D.E.F>	<.A.B.C.D>b

COMPLETION — Event

COMPLETION returns a bit string of length 1, specifying the completion value of x; the event can be active or inactive. If the completion value of the event is incomplete, '0'B is returned; if complete, '1'B is returned. The syntax for COMPLETION is:

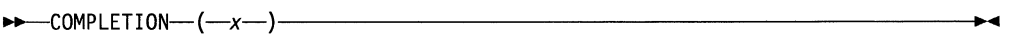


Abbreviation: CPLN

x Event reference.

COMPLETION — Pseudovariable

The pseudovariable sets the completion value of x. No interrupt can occur during each assignment to the pseudovariable. The COMPLETION pseudovariable cannot be used as the control variable in a do-specification. The syntax for COMPLETION pseudovariable is:

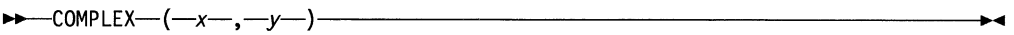


Abbreviation: CPLN

x Event reference. x must be inactive.

COMPLEX — Arithmetic

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:

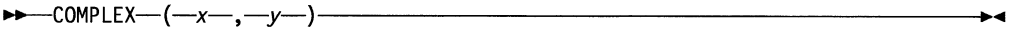
```
(MIN(N,MAX(p1-q1,p2-q2)+MAX(q1,q2)),
MAX(q1,q2))
```

where (p1,q1) and (p2,q2) are the precisions of x and y, respectively, and N is the maximum number of digits allowed.

If the arguments, after any necessary conversions have been performed, are floating point, the result has the precision of the longer argument.

COMPLEX — Pseudovariable

The pseudovariable assigns the real part of a complex value to x, and the real coefficient of the imaginary part to y. The attributes of x and y need not match, but if both are arrays they must have identical bounds. Only a complex value can be assigned to the pseudovariable. The COMPLEX pseudovariable cannot be used as the control variable in a do-specification. The syntax for COMPLEX is:



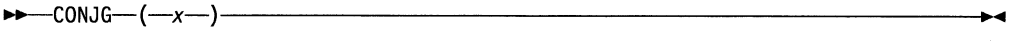
Abbreviation: CPLX

x and y
Real references.

Note: Use of COMPLEX pseudovariable should be avoided. Use IMAG or REAL instead.

CONJG — Arithmetic

CONJG returns the conjugate of x; that is, the value of the expression with the sign of the imaginary part reversed. The syntax for CONJG is:

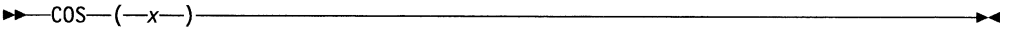


x Expression.

If x is real, it is converted to complex. The result has the base, scale, mode and precision of x.

COS — Mathematical

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:



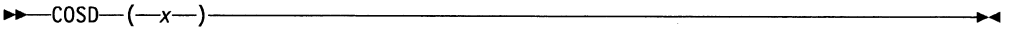
x Expression whose value is in radians.

If x = COMPLEX(a,b), the value of the result is given by:

```
COMPLEX(COS(a)*COSH(b),-SIN(a)*SINH(b))
```

COSD — Mathematical

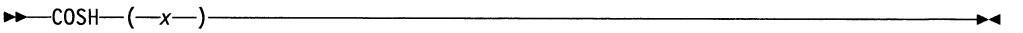
COSD returns a real floating-point value that has the base and precision of x, and is an approximation of the cosine of x. The syntax for COSD is:



x Real expression whose value is in degrees.

COSH — Mathematical

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:



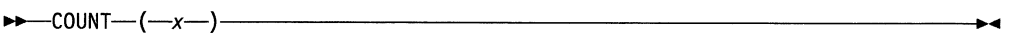
x Expression.

If x = COMPLEX(a,b), the value of the result is given by:

```
COMPLEX(COSH(a)*COS(b),SINH(a)*SIN(b))
```

COUNT — Input/output

COUNT returns a FIXED BINARY (15,0) value specifying the number of data items transmitted during the last GET or PUT operation on x. The syntax for COUNT is:



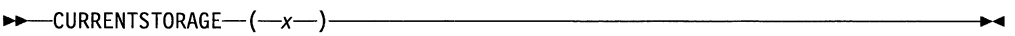
x File-reference. The file must be open and have the STREAM attribute.

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, 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.

CURRENTSTORAGE — Storage control

CURRENTSTORAGE returns a FIXED BINARY (31,0) value giving the implementation-defined storage, in bytes, required by x. The syntax for CURRENTSTORAGE is:



Abbreviation: CSTG

x A variable of any data type, data organization, and storage class except:

- A BASED, DEFINED, parameter, subscripted, or structure-base-element variable that is an unaligned fixed-length bit string.
- A minor structure 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).
- A major structure 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 CURRENTSTORAGE(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.

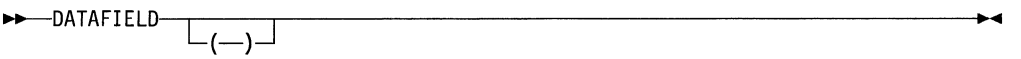
If x is an aggregate containing areas or varying-length strings, the returned value includes the area control bytes, the maximum sizes of the areas, the length prefixes of the strings, and the number of bytes in the maximum lengths of the strings. There is one *exception* to this rule:

If x is a structure 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.

CURRENTSTORAGE cannot be used to obtain the storage requirements of a structure mapped according to the COBOL mapping algorithm.

DATAFIELD — Condition-handling

DATAFIELD is in context in a NAME condition ON-unit (or any of its dynamic descendants), and 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:

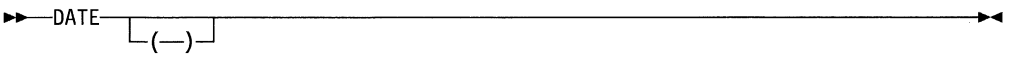


If the string that raised the condition contains DBCS identifiers, GRAPHIC data, or mixed character data, DATAFIELD returns a MIXED character string adjusted, if necessary, so the DBCS portions are enclosed in shift codes.

If DATAFIELD is used out of context, a null string is returned.

DATE — Miscellaneous

DATE returns a character string, length 6, in the format of yymmdd. The syntax for DATE is:



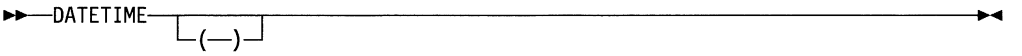
The returned character string represents:

yy Last two digits of the current year
mm Current month
dd Current day

The time zone and accuracy are system dependent.

DATETIME — Miscellaneous

DATETIME returns a character string, length 17, in the format of yyyymmddhhmmssttt. The syntax for DATETIME is:



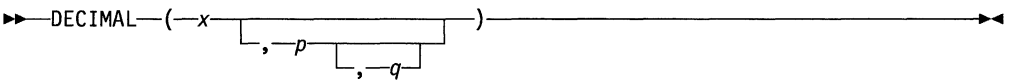
The returned character string represents:

yyyy Current year
mm Current month
dd Current day
hh Current hour
mm Current minute
ss Current second
ttt Current millisecond

The time zone and accuracy are system dependent.

DECIMAL — Arithmetic

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 Expression.

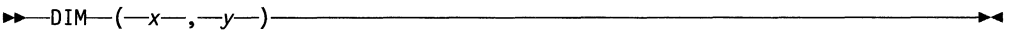
p Integer specifying the number of digits to be maintained throughout the operation; it must not exceed the implementation limit.

q Optionally-signed integer 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.

DIM — Array-handling

DIM returns a FIXED BINARY (31,0) value specifying the current extent of dimension y of x. The syntax for DIM is:



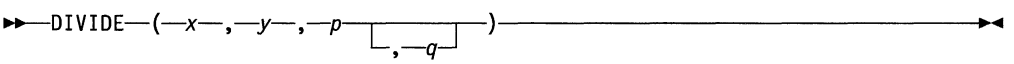
x Array expression. x must not have less than y dimensions, and x must not be an array of structures.

y Expression specifying a particular dimension of x. If necessary, y is converted to a FIXED BINARY (31,0) value. y must be greater than or equal to 1.

If the extent of an array dimension exceeds the allowable number for the implementation, the DIM function returns an undefined value.

DIVIDE — Arithmetic

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.

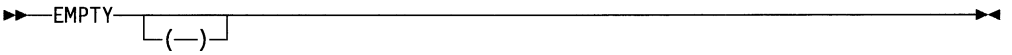
y Expression. If y=0, the ZERODIVIDE condition is raised.

p Integer specifying the number of digits to be maintained throughout the operation.

q Optionally-signed integer 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 — Storage control

EMPTY returns an area of zero extent. It can be used to free all allocations in an area. The syntax for EMPTY is:

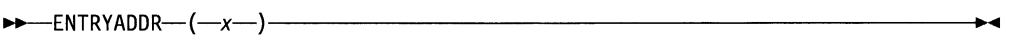


The value of this function is assigned to an area variable when the variable is allocated. For example:

```
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); */
```

ENTRYADDR — Storage control

ENTRYADDR returns a pointer value that is the address of the first executed instruction if the entry *x* is invoked. The entry *x* must be an external entry. The syntax for ENTRYADDR is:

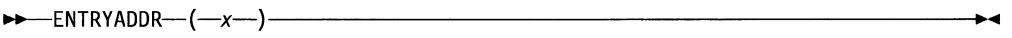


x Entry reference.

If x is a fetchable entry constant that has not been fetched or has been released, the NULL() value is returned.

ENTRYADDR — Pseudovariable

The ENTRYADDR pseudovariable initializes an entry variable, *x*, with the address of the entry to be invoked. The syntax for ENTRYADDR pseudovariable is:



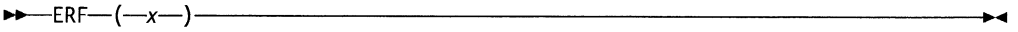
x Entry reference.

The ENTRYADDR pseudovariable cannot be used as the control variable in a DO loop.

Note: If the address supplied to the ENTRYADDR variable is the address of an internal procedure, unpredictable results might occur.

ERF — Mathematical

ERF returns a real floating-point value that is an approximation of the error function of x. The syntax for ERF is:



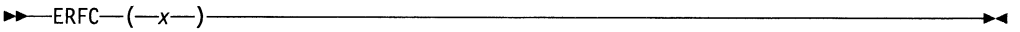
x Real expression.

The result has the base and precision of x, and a value given by:

$$(2/\mathrm{SQRT}(\mathrm{pi}))\int_0^x \mathrm{EXP}(-(t^{**}2))dt$$

ERFC — Mathematical

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:



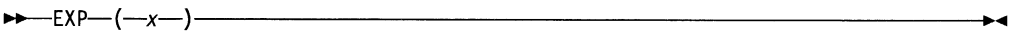
x Real expression.

The result has the base and precision of x, and a value given by:

```
1-ERF(x)
```

EXP — Mathematical

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:



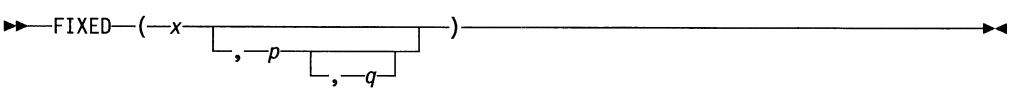
x Expression.

The result has the base, mode, and precision of x. If x=COMPLEX(a,b), the value of the result is given by:

```
(e**a)*COMPLEX(COS(b),SIN(b))
```

FIXED — Arithmetic

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.

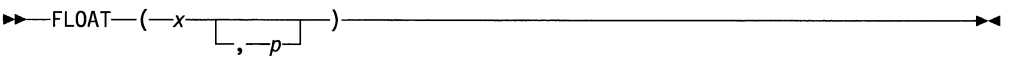
p Integer specifying the total number of digits in the result; it must not exceed the implementation limit.

q Optionally-signed integer specifying 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 default values (15,0) for a binary result, or (5,0) for a decimal result, are used.

FLOAT — Arithmetic

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:



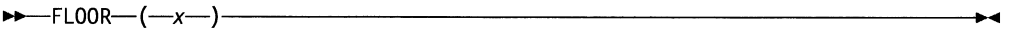
x Expression.

p Integer specifying the minimum number of digits in the result; it must not exceed the implementation limit.

If p is omitted, the default value 21, for a binary result, or 6, for a decimal result, is used.

FLOOR — Arithmetic

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)
```

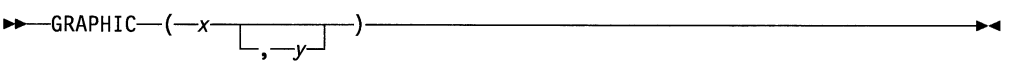
where N is the maximum number of digits allowed.

GRAPHIC — String-handling

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 GRAPHIC data type.

GRAPHIC returns the graphic value of x, with a length in graphic characters specified by y.

Characters convert to graphics, and shift codes, if any, are removed. 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 length change, with applicable padding or truncation. The CONVERSION condition is raised if either half of graphic character in the string contains a shift code.

When x is non-GRAPHIC, it is converted to character, if necessary. Each character is then converted to GRAPHIC. SBCS '40'X is converted to '4040'X, while other SBCS characters are converted by inserting '42'X in the first byte. Shift codes (if any) are discarded; DBCS data is copied. The CONVERSION condition is raised for any invalid use of shift codes, with the exception that the ERROR condition is raised for data ending with half a graphic or data with a missing shift-in.

y Expression

If necessary, y is converted to a FIXED BINARY (15,0) 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
123<.A.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:

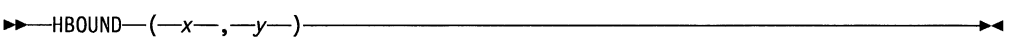
Conversion from CHARACTER to GRAPHIC, where the target is too short to contain the result.

```
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 — Array-handling

HBOUND returns a FIXED BINARY (31,0) value specifying the current upper bound of dimension y of x. The syntax for HBOUND is:

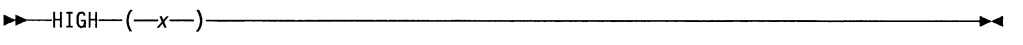


x Array expression. x must not have less than y dimensions, and x must not be an array of structures.

y Expression specifying a particular dimension of x. If necessary, y is converted to a FIXED BINARY (15,0) value. y must be greater than or equal to 1.

HIGH — String-handling

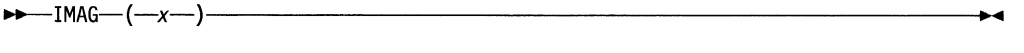
HIGH returns a character string of length x, where each character is the highest character in the collating sequence (hexadecimal FF). The syntax for HIGH is:



x Expression. If necessary, x is converted to a FIXED BINARY (15,0) value which must be positive. If x=0, the result is the null character string.

IMAG — Arithmetic

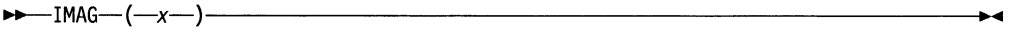
IMAG returns the coefficient of the imaginary part of x. The mode of the result is real, and the result has the base, scale and precision of x. The syntax for IMAG is:



x Expression. If x is real, it is converted to complex.

IMAG — Pseudovariable

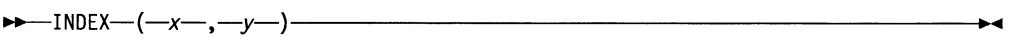
The pseudovariable assigns a real value or the real part of a complex value to the coefficient of the imaginary part of x. The syntax for IMAG is:



x Complex reference.

INDEX — String-handling

INDEX returns a FIXED BINARY (15,0) value indicating the starting position within x of a substring identical to y. The syntax for INDEX is:



x String-expression to be searched.

y String-expression to be searched for.

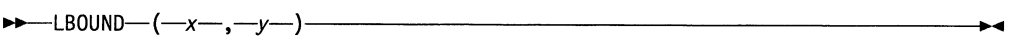
If y does not occur in x, or if either x or y have zero length, the value zero is returned.

If y occurs more than once in x, the starting position of the leftmost occurrence is returned.

If the first argument is GRAPHIC, the second must be GRAPHIC. If either argument is character or decimal, conversions are performed to produce character strings. Otherwise the arguments are bit and binary, or both binary, and conversions are performed to produce bit strings.

LBOUND — Array-handling

LBOUND returns a FIXED BINARY (31,0) value specifying the current lower bound of dimension y of x. The syntax for LBOUND is:

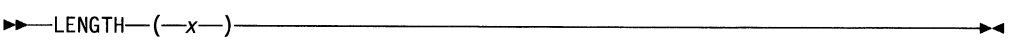


x Array expression. x must not have less than y dimensions, and x must not be an array of structures.

y Expression specifying the particular dimension of x. If necessary, y is converted to a FIXED BINARY (15,0) value. y must be greater than or equal to 1.

LENGTH — String-handling

LENGTH returns a FIXED BINARY (15,0) 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.

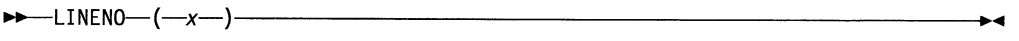
For example:

```
DECLARE A GRAPHIC(3);
```

The DECLARE statement defines 6 bytes of storage for A. Specifying LENGTH(A) returns the value 3.

LINENO — Input/output

LINENO returns a FIXED BINARY (15,0) value specifying the current line number of x. The syntax for LINENO is:

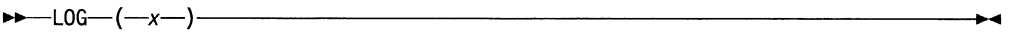


x File-reference.

The file must be open and have the PRINT attribute.

LOG — Mathematical

LOG returns a floating-point value that has the base, mode, and precision of x, and is an approximation of the natural logarithm (that is, the logarithm to the base e) of x. The syntax for LOG is:



x expression. If x is real, it must be greater than zero. If x is complex, it must not be equal to 0+0I.

The function is multiple-valued if x is complex; hence, only the principal value can be returned. The principal value has the form:

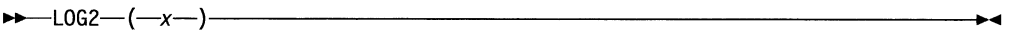
```
COMPLEX(a,b)
```

where a is nonnegative, and b is within the range:

```
-pi<b<=pi
```

LOG2 — Mathematical

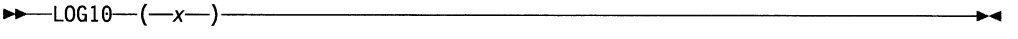
LOG2 returns a real floating-point value that has the base and precision of x, and is an approximation of the binary logarithm (that is, the logarithm to the base 2) of x. The syntax for LOG2 is:



x real expression that must be greater than zero.

LOG10 — Mathematical

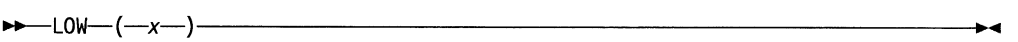
LOG10 returns a real floating-point value that has the base and precision of x, and is an approximation of the common logarithm (that is, the logarithm to the base 10) of x. The syntax for LOG10 is:



x real expression that must be greater than zero.

LOW — String-handling

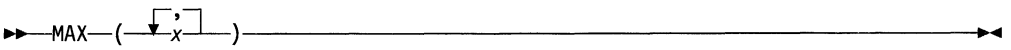
LOW returns a character string of length x, where each character is the lowest character in the collating sequence (hexadecimal 00). The syntax for LOW is:



x expression. If necessary, x is converted to a FIXED BINARY (15,0) value which must be positive. If x=0, the result is the null character string.

MAX — Arithmetic

MAX returns the largest value from a set of one or more expressions. The syntax for MAX is:



x expression.

All the arguments must be real; they convert to the common base and scale. 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:

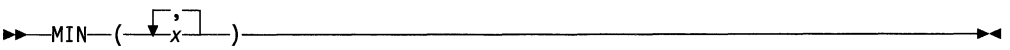
```
p1,p2,p3,...pn
```

then the precision of the result is given by:

```
MAX(p1,p2,p3,...pn)
```

MIN — Arithmetic

MIN returns the smallest value from a set of one or more expressions. The syntax for MIN is:



x expression.

All the arguments must be real; they will be converted to the common base and scale. The result is real with the common base and scale of the arguments.

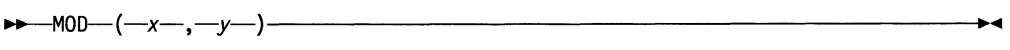
The precision of the result is the same as that described for the MAX built-in function, above.

MOD — Arithmetic

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.

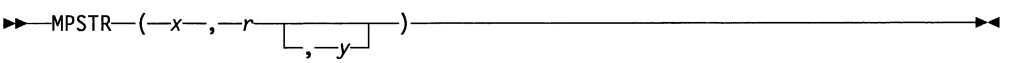
If x and y are fixed-point with different scaling factors, R might be truncated on the left, causing the SIZE condition to be raised.

For example:

```
MOD(10,8) is 2
MOD(-10,8) is 6
```

MPSTR — String-handling

MPSTR returns a mixed character string with balanced pairs of shift-out/shift-in codes. The processing of the string is determined by the rules selected by the expression **r**, as described below. The length of the returned string is equal to the length of the expression x, or to the value specified by y. The syntax for MPSTR is:



x expression that yields the character string result.

x cannot be GRAPHIC
x is converted to character if necessary

r expression that yields a character result.

r cannot be GRAPHIC
r is converted to character if necessary

r specifies the rules to be used for processing the string. The characters that can be used in r and their rules are as follows:

V or v Validates the mixed string x for balanced, unnested so/si pairs and returns a mixed string that has balanced pairs. V does not remove adjacent so/si pairs. If x contains unbalanced or nested so/si pairs, ERROR condition is raised.

S or s Removes adjacent so/si pairs and any null DBCS strings and creates a new string. Returns a mixed string with balanced so/si pairs.

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 FIXED BINARY (15,0) 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.

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:

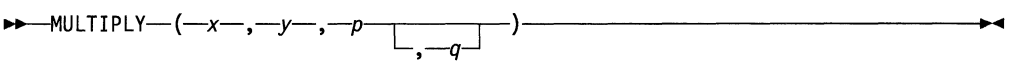
- Discarding excess characters from the right (if they are SBCS characters)

  or

- Discarding as many DBCS characters (2-byte pairs) as needed, then inserting a shift-in to make the DBCS data valid

MULTIPLY — Arithmetic

MULTIPLY returns x*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:



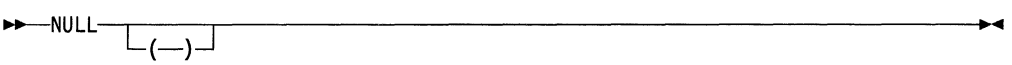
x and y
expressions.

p integer specifying the number of digits to be maintained throughout the operation; it must not exceed the implementation limit for the base and scale of the result.

q optionally-signed integer specifying 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 — Storage control

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:

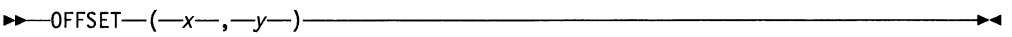


Note: NULL and SYSNULL do not compare equal. In this PL/I release, NULL returns the value 'FF000000'X and SYSNULL returns the value '00000000'X. However, you should not write code that depends on them being unequal.

See also "SYSNULL — Storage control" on page 401.

OFFSET — Storage control

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:



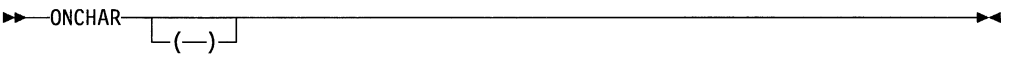
x pointer reference, which must identify a generation of a based variable within the area y, or be the null pointer value.

y area reference.

If x is an element reference, y must be an element variable.

ONCHAR — Condition-handling

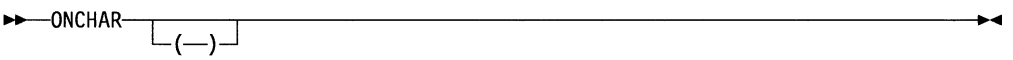
ONCHAR returns a character string of length 1, containing the character that caused the CONVERSION condition to be raised. It is in context in an ON-unit (or any of its dynamic descendants) for the CONVERSION condition or for the ERROR or FINISH condition raised as the implicit action for the CONVERSION condition. The syntax for ONCHAR is:



If the ONCHAR built-in function is used out of context a blank is returned, unless ONCHAR has a value given to it by an assignment to the pseudovariable out of context; in this case, the character assigned to the pseudovariable is returned by the built-in function.

ONCHAR — Pseudovariable

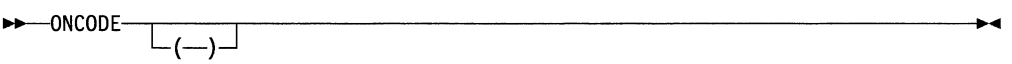
The pseudovariable sets the current value of the ONCHAR built-in function. The element value assigned to the pseudovariable is converted to a character value of length 1. The new character is used when the conversion is re-attempted. (See Chapter 15, "Conditions" on page 332, the "CONVERSION Condition" section). The syntax for ONCHAR pseudovariable is:



If the pseudovariable is used out of context, and the next reference to the built-in function is also out of context, the character assigned to the pseudovariable is returned. The out-of-context assignment is otherwise ignored.

ONCODE — Condition-handling

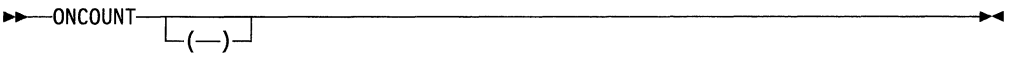
ONCODE returns a FIXED BINARY (15,0) value that is the condition code. It is in context in any ON-unit, or any dynamic descendant of an ON-unit. All condition codes are defined in Chapter 15, "Conditions." The syntax for ONCODE is:



If ONCODE is used out of context, zero is returned.

ONCOUNT — Condition-handling

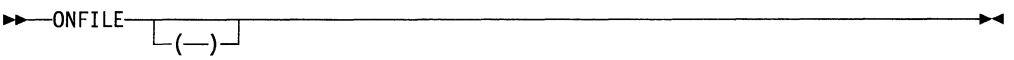
ONCOUNT returns a value specifying the number of conditions that remain to be handled when an ON-unit is entered. Multiple conditions are discussed under "Multiple conditions" on page 325. It is in context in any ON-unit, or any dynamic descendant of an ON-unit. The syntax for ONCOUNT is:



If ONCOUNT is used out of context, zero is returned.

ONFILE — Condition-handling

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 an ON-unit, or any of its dynamic descendants, for any input/output or CONVERSION condition, or 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.

ONKEY — Condition-handling

ONKEY returns a character string whose value is the key of the record that raised an input/output condition. For VSAM files, if the key is GRAPHIC, the string is returned as a mixed character string. It is in context in an ON-unit, or any of its dynamic descendants, for any input/output condition, except ENDFILE, or for 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:

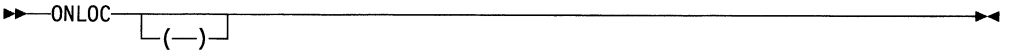
- For any input/output condition (other than ENDFILE), or for the ERROR or FINISH condition raised as implicit action for these conditions, the result is the value of the recorded key from the I/O statement causing the error.
- For REGIONAL(1) data sets, the result is a character string representation of the region 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 region 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 — Condition-handling

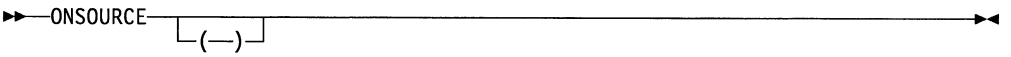
ONLOC returns a character string whose value is the name of the entry-point used for the current invocation of the procedure in which a condition was raised. If the name is a DBCS name, it is returned as a mixed character string. It is in context in any ON-unit, or in any of its dynamic descendants. The syntax for ONLOC is:



If ONLOC is used out of context, a null string is returned.

ONSOURCE — Condition-handling

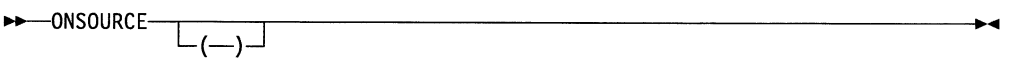
ONSOURCE returns a character string whose value is the contents of the field that was being processed when the CONVERSION condition was raised. It is in context in an ON-unit, or any of its dynamic descendants, for the CONVERSION condition or for the ERROR or FINISH condition raised as the implicit action 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 new string is used when the conversion is re-attempted. (See Chapter 15, the "CONVERSION Condition" section). 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 should 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.

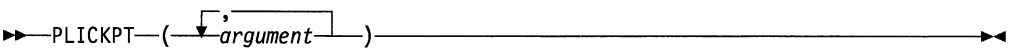
PLICANC — Built-in subroutine

In some implementations, it is possible to cancel any automatic restart activity from checkpoints taken in your program by calling the built-in subroutine PLICANC. However, VSE does not support this function. If your program calls PLICANC, the system will issue a message and the program will continue normal operation.

The syntax for PLICANC is:

PLICKPT — Built-in subroutine

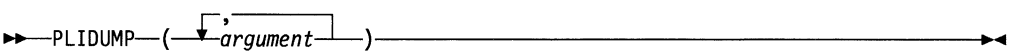
This built-in subroutine allows you to take a checkpoint for later restart. The syntax for PLICKPT is:



For more information, see the *PL/I VSE Programming Guide.*

PLIDUMP — Built-in subroutine

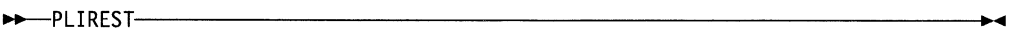
This built-in subroutine allows you to obtain a formatted dump of selected parts of storage used by your program. The syntax for PLIDUMP is:



For more information, see the *LE/VSE Debugging Guide and Run-Time Messages.*

PLIREST — Built-in subroutine

Automatic restart functions are not available under VSE. However, for compatibility with the MVS implementation of PL/I, you can code a call to the built-in subroutine PLIREST in your program. The syntax for PLIREST is:



For more information, see the *PL/I VSE Programming Guide.*

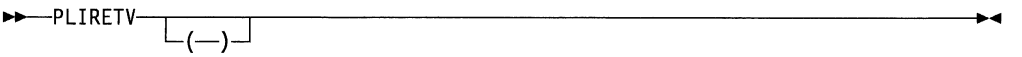
PLIRETC — Built-in subroutine

This built-in subroutine allows you to set a return code that can be examined by the program or (sub)system that invoked this PL/I program or by another PL/I procedure via the PLIRETV built-in function. The syntax for PLIRETC is:

For more information, see the *PL/I VSE Programming Guide* and *LE/VSE Programming Guide.*

PLIRETV — Miscellaneous

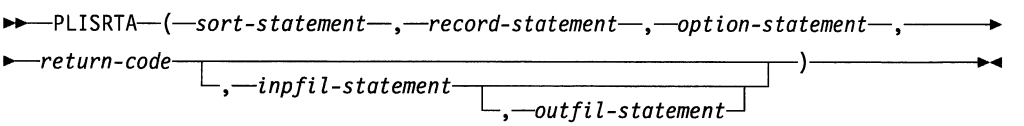
PLIRETV returns a FIXED BINARY (15,0) value that is the PL/I return code. The syntax for PLIRETV is:



The value of the PL/I return code is the most recent value specified by a CALL PLIRETC statement or the value returned by a COBOL or assembler routine whose entry point is declared with the option OPTIONS(RETCODE), or zero. For information about other services that can set the value returned in PLIRETV, see the *LE/VSE Programming Guide.*

PLISRTA — Built-in subroutine

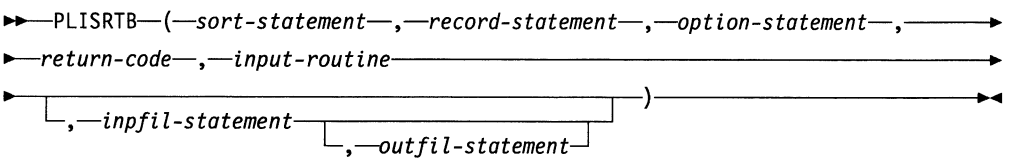
This built-in subroutine allows you to sort an input file to produce a sorted output file. The syntax for PLISRTA is:



For more information, see the *PL/I VSE Programming Guide.*

PLISRTB — Built-in subroutine

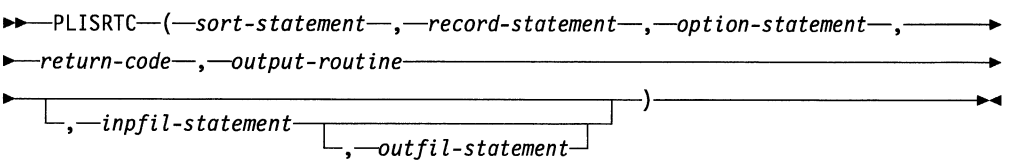
This built-in subroutine allows you to sort input records provided by an E15 PL/I exit procedure to produce a sorted output file. The syntax for PLISRTB is:



For more information, see the *PL/I VSE Programming Guide.*

PLISRTC — Built-in subroutine

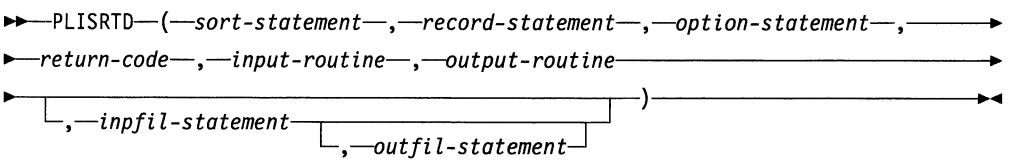
This built-in subroutine allows you to sort an input file to produce sorted records that are processed by an E35 PL/I exit procedure. The syntax for PLISRTC is:



For more information, see the *PL/I VSE Programming Guide.*

PLISRTD — Built-in subroutine

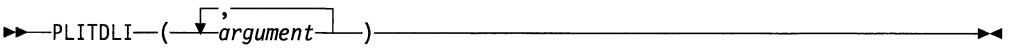
This built-in subroutine allows you to sort input records provided by an E15 PL/I exit procedure to produce sorted records that are processed by an E35 PL/I exit procedure. The syntax for PLISRTD is:



For more information, see the *PL/I VSE Programming Guide.*

PLITDLI — Subroutine

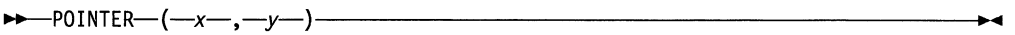
For a description of this subroutine and its arguments, refer to *DL/I DOS/VS Application Programming: CALL and RQDLI Interfaces.* The syntax for PLITDLI is:



PLITDLI cannot be declared with the BUILTIN attribute but is treated as a special subroutine.

POINTER — Storage control

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

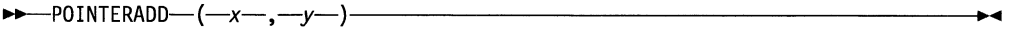
x offset reference, which can be the null offset value; if it is not, it must identify a generation of a based variable, but not necessarily in y. If it is not in y, the generation must be equivalent to a generation in y.

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 — Storage control

POINTERADD returns a pointer value that is the sum of its expressions. The syntax for POINTERADD is:



Abbreviation: PTRADD

x must be a pointer expression.

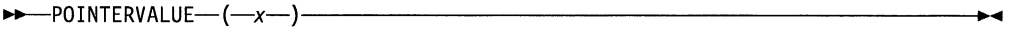
y must be a BIT, REAL FIXED BIN(p,0) or REAL FIXED DEC(p,0) expression. It will be converted to REAL FIXED BIN(31,0) if necessary.

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.

POINTERVALUE — Storage control

POINTERVALUE returns a pointer value that is the converted value of x. The syntax for POINTERVALUE is:



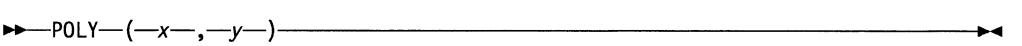
Abbreviation: PTRVALUE

x must be a BIT, REAL FIXED BIN(p,0) or REAL FIXED DEC(p,0) expression. It is converted to REAL FIXED BIN(31,0), if necessary.

POINTERVALUE(x) can be used to initialize static pointer variables if x is a constant.

POLY — Array-handling

POLY returns a floating-point value that is an approximation of a polynomial formed from two one-dimensional array expressions x and y. The returned value has a mode and base given by the rules for expression evaluation, and the precision of the longest argument. The syntax for POLY is:



x an array expression defined as x(m:n), where (m:n) represents the lower and upper bounds. If x is a reference to a defined variable, x must not be iSUB-defined.

y an array expression defined as y(a:b), where (a:b) represents the lower and upper bounds; or, an element-expression. If y is a reference to a defined variable, y must not be iSUB-defined.

If the arguments are not floating-point, they are converted to floating-point.

If m=n, the result is x(m). If m¬=n and y is an array, the value of the result is given by:

$$x(m) + \sum_{j=1}^{n-m} \left(x(m+j) * \prod_{i=0}^{j-1} y(a+i)\right)$$

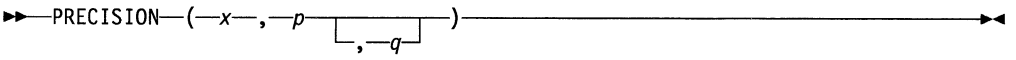
If (b-a)<(n-m-1) then y(a+i)=y(b) for (a+i)>b.

If m¬=n and y is an element-expression, it is interpreted as an array of one element, y(1), and the value of the result is given by:

$$\sum_{j=0}^{n-m} x(m+j)*y(1)**j$$

PRECISION — Arithmetic

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 integer specifying the number of digits that the value of the expression x is to have after conversion; it must not exceed the implementation limit for the base and scale.

q optionally-signed integer specifying 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.

PROD — Array-handling

PROD returns the product of all the elements in x. The syntax for PROD is:

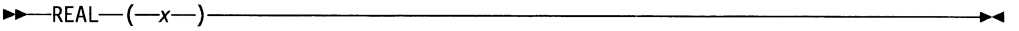
x array expression. If the elements of x are strings, they are converted to fixed-point integer values. If x is a reference to a defined variable, x must not be iSUB-defined.

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.

REAL — Arithmetic

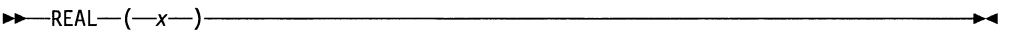
REAL returns the real part of x. The result has the base, scale, and precision of x. The syntax for REAL is:



x expression. If x is real, it is converted to complex.

REAL — Pseudovariable

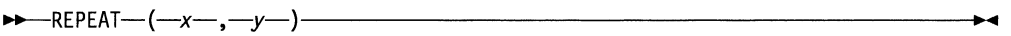
The pseudovariable assigns a real value or the real part of a complex value to the real part of x. The syntax for REAL pseudovariable is:



x complex reference.

REPEAT — String-handling

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, it is converted to bit string if it is binary, character string if it is decimal.

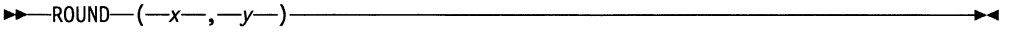
y expression. If necessary, y is converted to a FIXED BINARY (15,0) value.

If y is zero or negative, the string x is returned.

REPEAT does not support GRAPHIC data.

ROUND — Arithmetic

The value of x is rounded at a digit specified by y. The result has the mode, base, and scale of x. The syntax for ROUND is:



x expression.

y optionally-signed integer, specifying the digit at which rounding is to occur. Y must conform to the limits of scaling-factors for FIXED data. If y is greater than 0, it is the (y)th digit to the right of the point; if zero or negative, it is the (1-y)th digit to the left of the point. The valid range of Y is: $127 \geq y \geq -128$.

If x is floating-point, y must be specified but is ignored; the rightmost bit of the machine representation of the mantissa is set to 1, and the result has the precision of x.

If x is negative, the absolute value is rounded and the sign is restored.

The precision of a fixed-point result is given by:

```
(MAX(1,MIN(p-q+1+y,N)),y)
```

where (p,q) is the precision of x, and N is the maximum number of digits allowed. Thus y specifies the scaling factor of the result.

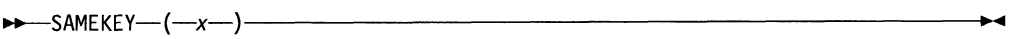
In the following example:

```
DCL X FIXED DEC(5,4) INIT(6.6666);
PUT (ROUND(X,2));
```

the value 6.67 is output.

SAMEKEY — Input/output

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.

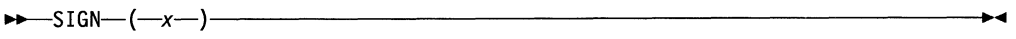
Upon successful completion of an input/output operation on file x, or immediately before the RECORD condition is raised, the value accessed by SAMEKEY is set to '1'B if the record processed is followed by another record with the same key, and set to '0'B if it is not.

The value accessed by SAMEKEY is also set to '0'B if:

- An input/output operation which raises a condition other than RECORD also causes file positioning to be changed or lost
- The file is not open.
- The file is not associated with a VSAM path accessing a data set through an alternate index.
- The record processed is not followed by another record with the same key

SIGN — Arithmetic

SIGN returns a FIXED BINARY (15,0) value that indicates whether x is positive, zero, or negative. The syntax for SIGN is:



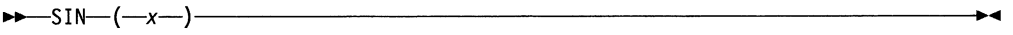
x real expression.

The returned value is given by:

Value of x	Value Returned
x > 0	+1
x = 0	0
x < 0	-1

SIN — Mathematical

SIN returns a floating-point value that has the base, mode, and precision of x, and is an approximation of the sine of x. The syntax for SIN is:



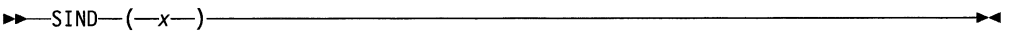
x expression whose value is in radians.

If x=COMPLEX(a,b), the value of the result is given by:

```
COMPLEX(SIN(a)*COSH(b),COS(a)*SINH(b))
```

SIND — Mathematical

SIND returns a real floating-point value that has the base and precision of x, and is an approximation of the sine of x. The syntax for SIND is:



x real expression whose value is in degrees.

SINH — Mathematical

SINH returns a floating-point value that has the base, mode, and precision of x, and represents an approximation of the hyperbolic sine of x. The syntax for SINH is:

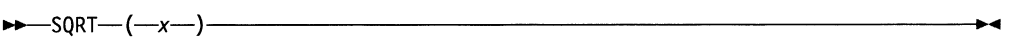
x expression whose value is in radians.

If x=COMPLEX(a,b), the value of the result is given by:

```
COMPLEX(SINH(a)*COS(b),COSH(a)*SIN(b))
```

SQRT — Mathematical

SQRT returns a floating-point value that has the base, mode, and precision of x, and is an approximation of the positive square root of x. The syntax for SQRT is:



x expression. If x is real, it must not be less than zero.

If x=COMPLEX(a,b), the value of the result is given by:

If x is complex, the function is multiple-valued; hence, only the principal value can be returned. The principal value has the form COMPLEX(a,b),

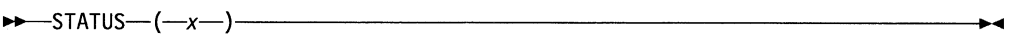
STATUS — Event

STATUS returns a FIXED BINARY (15,0) value specifying the status value of an event-reference x. If the event-variable is normal, zero is returned; if abnormal, nonzero is returned. The syntax for STATUS is:

x event-reference.

STATUS — Pseudovariable

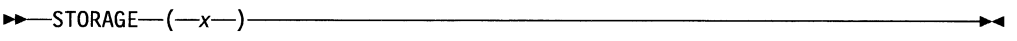
The pseudovariable sets the status value of an event-reference x. The variable can be active or inactive, and complete or incomplete. The value assigned to the pseudovariable is converted to FIXED BINARY (15,0), if necessary. No interrupt can occur during each assignment to the pseudovariable. The syntax for STATUS is:



x event-reference.

STORAGE — Storage control

STORAGE returns a FIXED BINARY (31,0) value giving the implementation-defined storage, in bytes, allocated to a variable x. The syntax for STORAGE is:



Abbreviation: STG

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-base-element variable that is an unaligned fixed-length bit string
- A minor structure 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)
- A major structure 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 STORAGE(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);
```

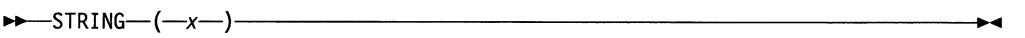
If x is:

- 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.

STORAGE cannot be used to obtain the storage requirements of a structure mapped according to the COBOL mapping algorithm.

STRING — String-handling

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. If x is a reference to a defined variable, x must not be iSUB-defined. Each base element of x must be either all bit-string, or all character-string and/or numeric character, in any combination.

If x is a structure that has padding caused by ALIGNED elements, the padding is not included in the result.

If any of the strings in the aggregate x are of varying length, only the current length, not including the 2-byte length prefix, is concatenated.

If x is an element variable, the rules for aggregates apply except that there is no concatenation.

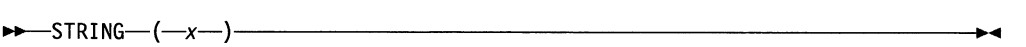
STRING — Pseudovariable

The pseudovariable assigns a bit or character expression, piece by piece, to x, until either all of the elements are filled or no piece of the assigned string remains. Any remaining strings in x are filled with blanks or zero bits, or, if varying-length, are given zero length.

The STRING pseudovariable must not be used in the data specification of a GET statement, in an INTO or KEYTO option of a READ statement, in the REPLY option of the DISPLAY statement, nor the KEYTO option of a WRITE statement.

The STRING pseudovariable cannot be used as the control variable in a do-specification.

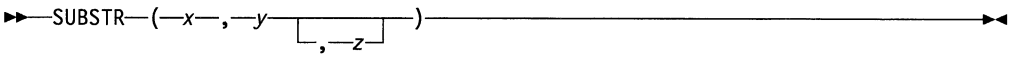
A varying-length string is filled to its maximum length, if possible. The syntax for STRING pseudovariable is:



x aggregate or element reference. If x is a reference to a defined variable, x must not be iSUB-defined. Each base element of x must be either all bit-string or all character-string.

SUBSTR — String-handling

SUBSTR returns a substring, specified by y and z, of x. The syntax for SUBSTR is:



x string-expression from which the substring is to be extracted. If x is not a string, it is converted to a bit string if binary, or a character string if decimal.

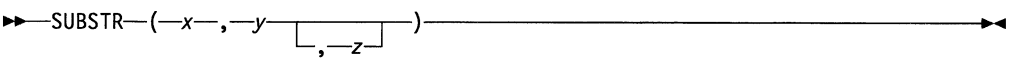
y expression that can be converted to a FIXED BINARY (15,0) value specifying the starting position of the substring in x.

z expression that can be converted to a value specifying 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 or z is omitted.

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.

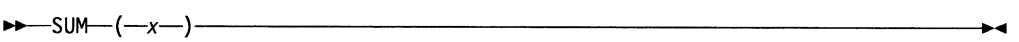
y expression that can be converted to a FIXED BINARY (15,0) value specifying the starting position of the substring in x.

z expression that can be converted to a FIXED BINARY (15,0) value specifying 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.

y and z can be arrays only if x is an array.

SUM — Array-handling

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:



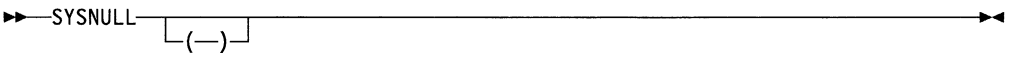
x array expression. If the elements of x are strings, they are converted to fixed-point integer values. If x is a reference to a defined variable, x must not be iSUB-defined.

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.

SYSNULL — Storage control

SYSNULL returns the system null pointer value. It can be used to initialize static pointer and offset variables. It also can be assigned or converted to offset variables (like NULL). The syntax for SYSNULL is:



SYSNULL is valid without the LANGLVL(SPROG) option. In the following example:

```
BINVALUE(SYSNULL())
```

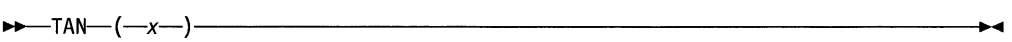
the returned value is zero.

Note: NULL and SYSNULL do not compare equal. In this PL/I release, NULL returns the value 'FF000000'X and SYSNULL returns the value '00000000'X. However, you should not write code that depends on them being unequal.

See also "NULL — Storage control" on page 386.

TAN — Mathematical

TAN returns a floating-point value that has the base, mode, and precision of x, and is an approximation of the tangent of x. The syntax for TAN is:



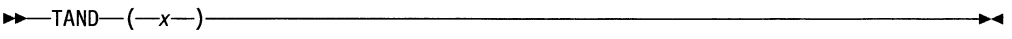
x expression whose value is in radians.

If x=COMPLEX(a,b), the value of the result is given by:

```
REAL(TAN(x)) = SIN(2*a)/
               (COS(2*a)+COSH(2*b))

IMAG(TAN(x)) = SINH(2*b)/
               (COS(2*a)+COSH(2*b))
```

TAND — Mathematical

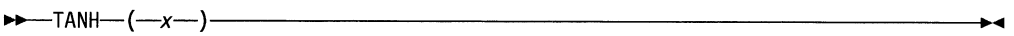
TAND returns a real floating-point value that has the base and precision of x, and is an approximation of the tangent of x. The syntax for TAND is:



x real expression whose value is in degrees.

TANH — Mathematical

TANH returns a floating-point value that has the base, mode, and precision of x, and is an approximation of the hyperbolic tangent of x. The syntax for TANH is:



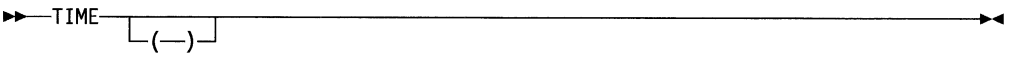
x expression whose value is in radians.

If x is complex, the value of the result is given by:

```
-1I*TAN(1I*x)
```

TIME — Miscellaneous

TIME returns a character string, length 9, in the format of hhmmssttt. The syntax for TIME is:



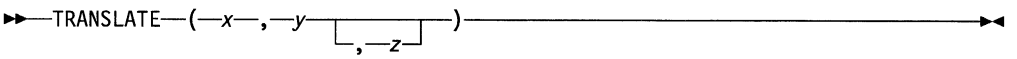
The returned character string represents:

hh current hour
mm current minute
ss current second
ttt current millisecond

The time zone and accuracy are system dependent.

TRANSLATE — String-handling

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, a string of 256 characters is assumed; it contains one instance of each EBCDIC code arranged in ascending collating sequence (hexadecimal 00 through FF).

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.

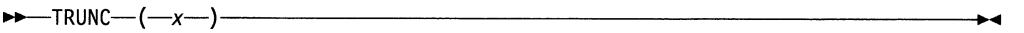
Any arithmetic or bit arguments are converted to character. For example:

```
DECLARE (W, X) CHAR (3);
X='ABC';
W = TRANSLATE (X, 'TAR', 'DAB');
/* W = 'ARC' */
```

TRANSLATE does not support GRAPHIC data.

TRUNC — Arithmetic

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.

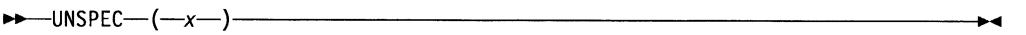
The base, mode, scale, and precision of the result match those of x, except when x is fixed-point with precision (p,q), the precision of the result is given by:

```
(MIN(N,MAX(p-q+1,1)),0)
```

where N is the maximum number of digits allowed.

UNSPEC — String-handling

UNSPEC returns a bit string that is the internal coded form of x. The syntax for UNSPEC is:



x expression.

Note: If you intend to migrate the program to OS/2, be aware that in PL/I Package 2, the UNSPEC of an array yields a result of BIT(*) scalar. If you need an array of returned values from UNSPEC, use a loop to obtain the result for each array element, and store each result in the array.

The length of the returned bit string depends on the attributes of x, as shown in Table 33.

Table 33 (Page 1 of 2). Length of bit string returned by UNSPEC

Bit-string length[1]	Attribute of x
16	FIXED BINARY (p,q), p<=15
32	FIXED BINARY (p,q), p>=16 FLOAT BINARY (p), p>=21 FLOAT DECIMAL (p), p<=6 OFFSET FILE constant or variable POINTER

[1] The bit-string lengths listed in this table are system dependent. The lengths listed are for the 370 system. Lengths are equal to 8 times the value given by the STORAGE built-in function.

Table 33 (Page 2 of 2). Length of bit string returned by UNSPEC

Bit-string length[1]	Attribute of x
64	FLOAT BINARY(p), 22<=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 POINTER
256	EVENT
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.
16+(16*n)	GRAPHIC VARYING where n is the maximum length of x.
8*(n+16)	AREA (n)
8*FLOOR(n)	FIXED DECIMAL (p,q) where n = (p+2)/2

[1] The bit-string lengths listed in this table are system dependent. The lengths listed are for the 370 system. Lengths are equal to 8 times the value given by the STORAGE built-in function.

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. If x is complex, the length of the returned string is twice the value given in Table 33.

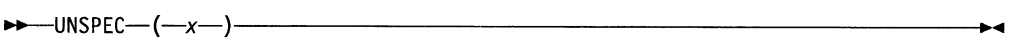
For example:

```
R = ARRAY(UNSPEC('A'));
```

In this statement, the internal representation of the character 'A' (a bit string 8 bits in length) is converted to a fixed binary arithmetic value, and used as a subscript for the array. (The decimal value of this particular subscript is 193).

UNSPEC — Pseudovariable

The 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 Table 33. If x is a complex variable, the length of the reference is twice that shown in Table 33. The syntax for UNSPEC pseudovariable is:

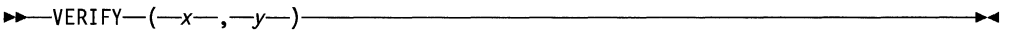


x reference.

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.

VERIFY — String-handling

VERIFY returns a FIXED BINARY (15,0) value indicating the position in x of the leftmost character or bit that is not in y. 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.

If either argument is character or decimal, conversions are performed to produce character strings. Otherwise, the arguments are bit and binary or both binary, and conversions are performed to produce bit strings.

In the following example, the VERIFY built-in function is used to test whether or not a string is all-numeric:

```
VERIFY (x,'0123456789')
```

The value is zero(0) if x is all-numeric; otherwise, it is nonzero.

VERIFY does not support GRAPHIC data.

Accuracy of mathematical built-in functions

Table 34 shows the accuracy of the mathematical built-in functions when using short- and long-precision floating-point arguments. Table 35 on page 409 shows the accuracy of the functions with extended-precision floating-point arguments.

Table 34 (Page 1 of 3). Accuracy of the mathematical built-in functions with short and long precision floating-point arguments

Function name	Argument mode	Range	Short floating point		Long floating point	
			Relative error *10**8		Relative error *10**17	
			RMS	MAX	RMS	MAX
ACOS(x)	real	ABS(x)<=0.5	43	88	7.2	20
		0.5<ABS(x)<=1	16	89	6.6	21
ASIN(x)	real	ABS(x)<=0.5	10	54	4.4	21
		0.5<ABS(x)<=1	26	94	5.9	21
ATAN(x)	real	ABS(x)<1	13	90	4.1	21
		full range[2]	25	99	5.2	17
	complex	full range[2]	21	110	5.2	44
ANTAN(x,y)	real	ABS(x)<=1, ABS(y)<1[2]	29	160	6.9	36
ATAND(x)	Similar to real ATAN(x)					
ATANH(x)	real	ABS(x)<=0.2	46	110	-	-
		ABS(x)<=0.9	39	120	-	-
		ABS(x)<=0.25	-	-	5.8	21
		ABS(x)<0.95	-	-	9.0	25
	complex	full range[2]	22	120	5.6	41
COS(x)	real[1]	0<=x<=pi	4.7	12	7.3	27
		-10<=x<0,pi<x<=10	4.6	12	6.9	27
		10<ABS(x)<=100	4.6	12	100	270
	complex[3]	ABS(a)<=10,ABS(b)<=1	120	320	31	380
COSD(x)	similar to real COS(x)					
COSH(x)	real	ABS(x)<=1	41	96	-	-
		1<ABS(x)<2	21	72	-	-
		ABS(x)<=170	20	82	-	-
		ABS(x)<=17	-	-	11	39
		ABS(x)<=5	-	-	11	38
	complex[3]	ABS(a)<=10, ABS(b)<=1	97	310	25	73
ERF(x)	real	ABS(x)<=1	11	85	2.6	19
		1<ABS(x)<2.04	3.7	11	0.95	2.9
		2.04<ABS(x) <3.9192	3.5	6.0	-	-
		2.04<ABS(x) <6.092	-	-	0.80	1.4

Table 34 (Page 2 of 3). Accuracy of the mathematical built-in functions with short and long precision floating-point arguments

Function name	Argument mode	Range	Short floating point		Long floating point	
			Relative error *10**8		Relative error *10**17	
			RMS	MAX	RMS	MAX
ERFC(x)	real	-3.8<x<0	30	94	-	-
		-6<x<0	-	-	6.5	21
		0<=x<=1	13	69	2.7	15
		1<x<=2.04	37	200	9.1	43
		2.04<x<4	37	130	8.7	33
		4≤x<13.3	820	1500	200	350
EXP(x)	real	-1<x<1	13	44	5.4	21
		full range	12	46	4.7	43
	complex[3]	ABS(a)<=170 ABS(b)<pi/2	65	240	-	-
		ABS(a)<=170, pi/2<=ABS(b)<=20	63	230	-	-
		ABS(a)<1 ABS(b)<pi/2	-	-	19	62
		ABS(a)<20 ABS(b)<20	-	-	20	82
LOG(x)	real	excluding 0.5<x<2.0[2]	12	84	5.5	34
		0.5<x<2.0[1]	2.5	6.8	2.4	4.7
	complex	full range[2]	38	190	13	53
LOG2(x)	real	excluding 0.5<x<2.0[2]	34	98	8.8	43
		0.5<x<2.0[1]	23	48	2.9	5.8
LOG10(x)	real	excluding 0.5<x<2.0[2]	22	110	6.6	32
		0.5<x<2.0[1]	2.3	7.2	1.2	2.9
SIN(x)	real[1]	ABS(x)<=pi/2	4.8	12	1.8	7.7
		pi/2<ABS(x)<=10	4.6	13	32	240
		10<ABS(x)<=100	4.6	12	93	270
	complex[3]	ABS(a)<=10,ABS(b)<=1	120	340	200	11000
SIND(x)	Similar to real SIN(x)					
SINH(x)	real	ABS(x)<=1	20	88	-	-
		-1<ABS(x)<2	25	100	-	-
		ABS(x)<=170	20	82	-	-
		ABS(x)<=17	-	-	10	36
		ABS(x)<0.881374	-	-	3.7	20
		0.881374<ABS(x)<=5	-	-	10	35
	complex[3]	ABS(a)<=10,ABS(b)<=1	88	270	23	64
SQRT(x)	real	full range[2]	13	48	3.1	11
	complex	full range[2]	54	220	13	49

Table 34 (Page 3 of 3). Accuracy of the mathematical built-in functions with short and long precision floating-point arguments

Function name	Argument mode	Range	Short floating point		Long floating point	
			Relative error *10**8		Relative error *10**17	
			RMS	MAX	RMS	MAX
TAN(x)	real[4]	ABS(x)<=pi/4	29	160	6.2	39
		pi/4<ABS(x)<pi/2	37	150	-	-
		Pi/4<ABS(x)<1.5	-	-	47	230
		pi/2<ABS(x)<=10	32	480	-	-
		1.5<ABS(x)<=10	-	-	7800	47000
		10<ABS(x).100	31	140	7800	27000
	complex[3]	ABS(a)<=1,ABS(b)<=9	53	290	17	71
TAND(x)	Similar to real TAN(x)					
TANH(x)	real	ABS(x)<=0.7	15	78	-	-
		0.7<ABS(x)<=9.011	3.9	2.3	-	-
		ABS(x)<=0.54931	-	-	3.8	19
		0.54931<ABS(x) <=20.101	-	-	1.0	16
	complex[3]	ABS(a)<9,ABS(b)<1	52	270	17	69

Notes:

1. RMS and MAX values given are absolute errors.
2. All these ranges are distributed exponentially; all other distributions are uniform.
3. Where x=Complex(a,b)
4. Each figure here depends on the particular points encountered near the singularities of the function, where no error control can be maintained.

Table 35 (Page 1 of 2). Accuracy of the mathematical built-in functions with extended-precision floating-point arguments

Function name	Argument mode	Range	Distribution type	Relative error *10**34	
				RMS	Max
ACOS(x)	real	ABS(x)<=1	U	9.9	32
ASIN(x)	real	ABS(x)<=1	U	8.1	32
	real	ABS(x)<10**75	T	7.3	30
ATAN(x)	complex	full range	EU	12	170
ATN(x,y)	real	full range	EU	8.5	38
ATAND(x)	similar to real ATAN(x)				
		ABS(x)<0.25	U	8.6	28
	real	ABS(x)<=0.95	U	18	50
ATANH(x)	complex	full range	EU	11	59
		0<=x<pi	U	1.5	3.3
		−10<x<0,pi<=x<10	U	1.6	3.5
	real[1]	10<=ABS(x)<200	U	1.6	3.5
		ABS(a)<10	U		
COS(x)	complex[2]	ABS(b)<1	U	24	62
COSD(x)	similar to real COS(x)				
	real	ABS(x)<10	U	15	61
		ABS(a)<10	U		
COSH(x)	complex[2]	ABS(b)<1	U	20	67
		ABS(x)x<1	U	5.3	30
		1<=ABS(x)<2.8437	U	2.3	9.2
ERF(x)	real	2.8437<=ABS(x)<5	U	1.3	1.9
		−5<x<0	U	12	31
		0<=x<1	U	5.8	33
		1<=x<2.8437	U	28	77
ERFC(x)	real	2.8437<=x<5	U	180	490
		ABS(x)<1	U	4.3	15
		ABS(x)<10	U	3.8	15
	real	−180<x<174	U	3.7	15
		ABS(a)<170	U		
		ABS(b)<pi/2	U	7.8	35
		ABS(a)<170	U		
EXP(x)	complex[2]	pi/2<=ABS(b)<100	U	8.0	33
		0.99<x<1.0[1]	U	0.084	0.20
		0.5<x<2sup1.	U	1.7	3.2
	real	10**−78<x<10**75	E	8.9	45
LOG(x)	complex	full range	EU	9.8	51
		0.99<x<1.01[1]	U	0.055	0.13
		0.5<x<2[1]	U	1.0	1.9
LOG2(x)	real	10**−78<x<10**75	E	4.4	30
		0.99<x<1.01[1]	U	0.038	0.16
		0.5<x<2[1]	U	1.5	2.9
LOG10(x)	real	10**−78<x<10**75	E	12	38

Table 35 (Page 2 of 2). Accuracy of the mathematical built-in functions with extended-precision floating-point arguments

Function name	Argument mode	Range	Distribution type	Relative error *10**34	
				RMS	Max
		ABS(x)<pi/2	U	1.2	3.0
		pi/2<=ABS(x)<10	U	1.6	3.5
	real[1]	10<=ABS(x)<200	U	1.5	3.6
		ABS(a)<10	U		
SIN(x)	complex[2]	ABS(b)<1	U	24	60
SIND(x)	similar to real SIN(x)				
		ABS(x)<1	U	6.8	29
	real	1<=ABS(x)<10	U	13	54
		ABS(a)<10	U		
SINH(x)	complex[2]	ABS(b)<1	U	18	53
		10**–50<x<10**50	E	3.0	15
	real	10**–78<x<10**75	E	2.8	14
SQRT(x)	complex	full range	EU	7.1	21
		ABS(x)<pi/4	U	9.6	36
		pi/4<=ABS(x)<pi/2	U	8.9	39
		pi/2<=ABS(x)<10	U	12	52
	real	10<=ABS(x)<200	U	11	46
		ABS(a)<1	U		
TAN(x)	complex[2]	ABS(b)<9	U	15	61
TAND(x)	similar to real TAN(x)				
		ABS(x)<0.54931	U	5.0	25
	real	0.54931.ABS(x)<5	U	2.6	21
		ABS(a)<9	U		
TANH(x)	complex[2]	ABS(b)<1	U	14	53

Notes:

1. RMS and Max values are for absolute errors
2. Where X=COMPLEX(a,b)
3. E exponential
   EU x=r*EXP (k*1I), or (ATAN only)
   COMPLEX(x,y)=r*EXP(k*1I), and:
   r has E distribution in (0,10**75)
   k has U distribution in (–pi,pi)
   U uniform (linear)
   T tangents of linearly–scaled
   angles in (–pi/2,pi/2)

Chapter 17. Preprocessor facilities

Chapter 17. Preprocessor facilities

The compiler provides a preprocessor (a macroprocessor) for source program alteration. It is executed prior to compilation, when you specify the MACRO compiler option. The preprocessor scans the preprocessor input and generates preprocessor output. The preprocessor output can serve as input to the compiler.

This description of the preprocessor assumes that you know the PL/I language described throughout this publication.

Preprocessor input is a string of characters, graphics, or both consisting of intermixed:

- *Preprocessor statements.*[1]

  Preprocessor statements are executed as they are encountered by the preprocessor scan (with the exception of preprocessor procedures, which must be invoked in order to be executed). Preprocessor statements, except those in preprocessor procedures, begin with a percent symbol (%). Using a blank to separate the percent symbol from the rest of the statement is optional.

  The preprocessor executes preprocessor statements and alters the input text accordingly. Preprocessor statements can cause alteration of the input text in any of the following ways:

  - Any identifier (and an optional argument list) appearing in the input text can be changed to an arbitrary string of text.
  - You can indicate which portions of the input text to copy into the preprocessor output.
  - A string of characters residing in a library can be included in the preprocessor input.
- *Listing control statements*, which control the layout of the printed listing of the program. These statements affect both the insource listing (the preprocessor input) and the source listing (the preprocessor output) and are described in Chapter 7, "Statements" on page 169.
- *Input text*, which is preprocessor input that is not a preprocessor statement or a listing control statement. The input text can be a PL/I source program or any other text, provided that it is consistent with the processing of the input text by the preprocessor scan, described below.

Preprocessor output[2] is a string of characters consisting of intermixed:

- *Listing control statements.* Listing control statements that are scanned in the preprocessor input are copied to the preprocessor output.
- *Output text.* Input text that is scanned and possibly altered is placed in the preprocessor output.

---

[1] For clarity in this discussion, preprocessor statements are shown with the % symbol (even though, when used in a preprocessor procedure, such a statement would not have a % symbol).

[2] Preprocessor replacement output is shown in a formatted style, while actual execution-generated replacement output is unformatted.

You can specify compiler options that cause the preprocessor input to be printed, and the preprocessor output to be printed, written, or both to a data set. The listing of the preprocessor input is the *insource listing*, and the listing of the preprocessor output is the *source listing*.

The GRAPHIC compiler option must be specified when the preprocessor input contains DBCS or mixed data. The rules for using DBCS and mixed data as input for the preprocessor are the same as for PL/I source input. Preprocessor identifiers can include DBCS and follow the same syntax rules as PL/I identifiers. Like the compiler, preprocessor statements can be written in SBCS, DBCS EBCDIC, or in both.

Preprocessor scan

The preprocessor starts its scan at the beginning of the preprocessor input and scans each character sequentially. It converts lowercase characters in the input (except for those in comments and string constants) to uppercase. It also converts all EBCDIC DBCS elements to SBCS elements. Preprocessor actions are as follows, for:

Preprocessor statements: Preprocessor statements are executed when encountered. You can:

- Define preprocessor names using the %DECLARE statement and appearance as a label prefix.

  If a preprocessor variable is not explicitly declared, it is an error and is diagnosed and the variable given the default attribute of CHARACTER. However, the variable is not activated for replacement unless it appears in a subsequently-executed %ACTIVATE statement. The variable can be referenced in preprocessor statements.
- Activate an identifier using the %DECLARE or %ACTIVATE statement, thus initiating replacement activity, as described below under "Input text" on page 414.
- Deactivate an identifier using the %DEACTIVATE statement, thus terminating replacement activity.
- Generate a message in the compiler listing using the %NOTE statement.
- Include string of characters into the preprocessor input.
- Cause the preprocessor to continue the scan at a different point in the preprocessor input using the %GOTO, %IF, %null, %DO, or %END statement.
- Change values of preprocessor variables using the %assignment or %DO statement.
- Define preprocessor procedures using the %PROCEDURE, %RETURN, and %END statements. A preprocessor procedure can be invoked by a function reference in a preprocessor expression, or, if the function procedure name is active, by encountering a function reference in the preprocessor scan of input text.

Listing control statements: Listing control statements that are not contained in a preprocessor procedure are copied into the preprocessor output, each on a line of its own.

Input text: The input text, after replacement of any active identifiers by new values, is copied into the preprocessor output. Invalid characters (part of a character constant or comment) are replaced with blanks in the preprocessor output. To determine replacements, the input text is scanned for:

- Characters that are not part of this PL/I character set are treated as delimiters and are otherwise copied to this output unchanged.
- PL/I character constants or PL/I comments. These are passed through unchanged from input text to preprocessor output by the preprocessor unless they appear in an argument list to an active preprocessor procedure. However, this can cause mismatches between input and output lines for strings or comments extending over several lines, when the input and output margins are different. The output line numbering in these cases also shows this mismatch.

  With the preprocessor, the hexadecimal values `'00'X` through `'06'X` must not be used. The values `'0E'X` and `'0F'X` are interpreted as control (shift-out, shift-in) codes when GRAPHIC compiler option is in effect.
- Active Identifiers. For an identifier to be replaced by a new value, the identifier must be first *activated* for replacement. Initially, an identifier can be activated by its appearance in a %DECLARE statement. It can be deactivated by executing a %DEACTIVATE statement, and it can be reactivated by executing a %ACTIVATE or %DECLARE statement.

  An identifier that matches the name of an active preprocessor variable is replaced in the preprocessor output by the value of the variable.

  When an identifier matches the name of an active preprocessor function (either programmer-written or built-in) the procedure is invoked and the invocation is replaced by the returned value.

  Identifiers can be activated with either the RESCAN or the NORESCAN options. If the NORESCAN option applies, the value is immediately inserted into the preprocessor output. If the RESCAN option applies, a rescan is made during which the value is tested to determine whether it, or any part of it, should be replaced by another value. If it cannot be replaced, it is inserted into the preprocessor output; if it can be replaced, replacement activity continues until no further replacements can be made. Thus, insertion of a value into the preprocessor output takes place only after all possible replacements have been made.

Replacement values must not contain % symbols, unmatched quotation marks, or unmatched comment delimiters.

The scan terminates when an attempt is made to scan beyond the last character in the preprocessor input. The preprocessor output is then complete and compilation can begin.

Preprocessor variables and data elements

A preprocessor variable is specified in a %DECLARE statement with either the FIXED or the CHARACTER attribute. No other attributes can be declared for a preprocessor variable. (Other attributes are supplied by the preprocessor, however.) All variables have storage equivalent to the STATIC storage class.

Preprocessor data types are coded arithmetic and string data, and are either:

FIXED DECIMAL (5,0)
A preprocessor variable declared with the FIXED attribute is given the attributes DECIMAL and precision (5,0). Fixed decimal constants must be integers.

CHARACTER varying
A preprocessor variable declared with the CHARACTER attribute is given the varying attribute. String repetition factors are not allowed for character constants.

Using mixed data in a preprocessor CHARACTER variable allows the preprocessor to construct and return DBCS or mixed data. When the variable uses mixed data, you must specify GRAPHIC as a compile-time option. The preprocessor observes continuation conventions, but it does not verify the validity of the returned values.

The preprocessor also supports the X character string constants. However, string repetition factors are not allowed.

BIT There are no preprocessor bit variables. However, bit constants are allowed, and bit values result from comparison operators, the concatenation operator (when used with bit operands), the *not* operator, and the PARMSET built-in function. The preprocessor-expression in the %IF statement converts to a bit value.

Preprocessor references and expressions

Preprocessor references and expressions are written and evaluated in the same way as described in Chapter 3, "Expressions and references," with the following additional comments:

- The operands of a preprocessor expression can consist only of preprocessor variables, references to preprocessor procedures, fixed decimal constants, bit constants, character constants, and references to preprocessor built-in functions.
- The exponentiation symbol (**) cannot be used.
- For arithmetic operations, only decimal arithmetic of precision (5,0) is performed; that is, each operand is converted to a decimal fixed-point integer value of precision (5,0) before the operation is performed, and the decimal fixed-point result is converted to precision (5,0). For example, the expression 3/5 evaluates to 0, rather than to 0.6.

  Any character value being converted to an arithmetic value must be in the form of an optionally-signed integer. A null string converts to 0.
- The conversion of a fixed-point value to a bit value always results in a string of length CEIL(3.32*5), that is, 17.
- The conversion of a fixed-point decimal value to a character value always results in a string of length 8. Leading zeros are replaced by blanks and an additional three blanks are appended to the left end; the rightmost blank is replaced by a minus sign if the value is negative.

Scope of preprocessor names

The scope of a preprocessor name is determined by where it is declared. The scope of a name declared within a preprocessor procedure is that procedure. The scope of a name declared within an included string is that string and all input text scanned after that string is included (except any preprocessor procedure in which the name is also declared). The scope of any other name is the entire preprocessor input (except any preprocessor procedure in which the name is also declared).

Preprocessor procedures

Preprocessor procedures are function procedures. A preprocessor procedure is delimited by %PROCEDURE and %END statements, and contains at least one RETURN statement.

The statements and groups that can be used within a preprocessor procedure are:

- The preprocessor assignment statement.
- The preprocessor DECLARE statement.
- The preprocessor do-group.
- The preprocessor GO TO statement. (A GO TO statement appearing in a preprocessor procedure cannot transfer control to a point outside of that procedure.)
- The preprocessor IF statement.
- The preprocessor null statement.
- The preprocessor RETURN statement.
- The preprocessor NOTE statement.
- The %PAGE, %SKIP, %PRINT, and %NOPRINT listing control statements.

Preprocessor statements in a preprocessor procedure do not begin with a percent symbol.

Preprocessor procedures cannot be nested. A preprocessor ENTRY cannot be in a preprocessor procedure.

A preprocessor procedure entry name, together with the arguments to the procedure, is called a *function reference*. A preprocessor procedure can be invoked by a function reference in a preprocessor expression, or, if the function procedure name is active, by encountering a function reference in the preprocessor scan of input text. Preprocessor procedure entry names need not be specified in %DECLARE statements.

Provided its entry name is active, a preprocessor procedure need not be scanned before it is invoked. It must, however, be present either in:

- The preprocessor input
- A string included prior to the point of invocation

The result of a preprocessor procedure reference encountered before that procedure is incorporated into the preprocessor input is undefined.

The value returned by a preprocessor function (that is, the value of the preprocessor expression in the RETURN statement) replaces the function reference and its associated argument list in the preprocessor output.

Arguments and parameters for preprocessor functions

The number of arguments in the procedure reference and the number of parameters in the %PROCEDURE statement need not be the same. The arguments are evaluated before any match is made with the parameter list. If there are more positional arguments than parameters, the excess arguments on the right are ignored. (For an argument that is a function reference, the function is invoked and executed, even if the argument is ignored later.) Parameters that are not set by the function reference are given values of zero, for FIXED parameters, or the null string, for CHARACTER parameters.

Parameters should not be set more than once by a function reference. However, if the value of a parameter is specified more than once, for example both by its position and by keyword, the error is diagnosed and the leftmost setting is used for the invocation.

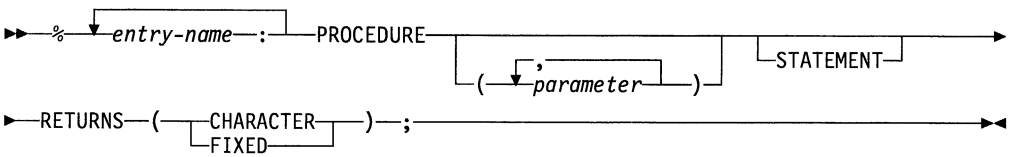
If the function reference appears in a preprocessor statement, the arguments are associated with the parameters in the normal fashion. Dummy arguments can be created and the arguments converted to the attributes of the corresponding parameters, in the same manner as described under "Association of arguments and parameters" on page 128.

If the function reference appears in input text, dummy arguments are always created. The arguments are interpreted as character strings and are delimited by a comma or right parenthesis. A comma or right parenthesis does not act as a delimiter, however, if it appears between matching parentheses, single quotes, or comment delimiters. For example, the positional argument list (A(B,C),D) has two arguments, namely, the string A(B,C) and the string D. Blanks in arguments (including leading and trailing blanks) are significant but, if such blanks extend to the end of a line and are not enclosed in quotes or comment delimiters, they are replaced by one blank.

When a function reference is encountered in input text, each argument is scanned for possible replacement activity. This replacement activity has no effect on the number of arguments passed to the function. Any commas or parentheses introduced into arguments by replacement activity are not treated as delimiters, but simply as characters in the argument. If keyword invocation is used, the keywords themselves are not eligible for replacement activity. After all replacements are made, each resulting argument is converted to the type indicated by the corresponding parameter attribute in the preprocessor procedure statement for the function entry name.

%PROCEDURE statement

The %PROCEDURE statement is used in conjunction with a %END statement to delimit a preprocessor procedure. The syntax for the %PROCEDURE statement is:



Abbreviation: %PROC

parameter
specifies a parameter of the function procedure.

STATEMENT
If the reference occurs in input text and the STATEMENT option is present:

- The arguments can be specified either in the positional argument list or by keyword reference.
- The end of the reference must be indicated by a semicolon. The semicolon is not retained when the replacement takes place.

For example, a preprocessor procedure headed by:

```
%FIND:PROC(A,B,C) STATEMENT...;
```

must be invoked from a preprocessor expression by a reference of the form:

```
FIND(arg1,arg2,arg3)
```

If the reference is in input text, the procedure can be invoked by any of the following references (or similar ones), all of which have the same result:

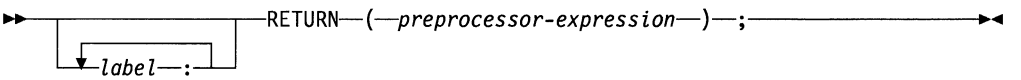
```
FIND(X,Y,Z);

FIND B(Y) C(Z) A(X);

FIND(X) C(Z) B(Y);

FIND(,Y,Z) A(X);
```

RETURNS
The attribute CHARACTER or FIXED must be specified in the RETURNS attribute list to specify the attribute of the value returned by the function procedure.

Preprocessor RETURN statement

The preprocessor RETURN statement can be used only in a preprocessor procedure and, therefore, can have no leading %. It returns a value as well as control back to the point from which the preprocessor procedure was invoked. At least one RETURN statement must appear in each preprocessor procedure.

The value returned by a preprocessor function procedure to the point of invocation is specified by the preprocessor-expression in a RETURN statement in the procedure. The syntax of the preprocessor RETURN statement is:



preprocessor-expression
The value is converted to the RETURNS attribute specified in the %PROCEDURE statement before it is passed back to the point of invocation.

Preprocessor built-in functions

A function reference can invoke one of a set of predefined functions called *preprocessor built-in functions*. These built-in functions are invoked in the same way that programmer-defined functions are invoked, except that they must be invoked with the correct number of arguments.

The preprocessor built-in functions are:

```
COMPILETIME        LENGTH
COUNTER            PARMSET
INDEX              SUBSTR
```

The preprocessor executes a reference to a preprocessor built-in function in input text only if the built-in function name is active. The built-in functions can be activated by a %DECLARE or %ACTIVATE statement.

In preprocessor statements, the preprocessor built-in function names are always active as built-in functions unless they are declared with some other meaning.

If a preprocessor built-in function name is used as the name of a user-defined preprocessor procedure, references to the name are references to the procedure, not to the built-in function. In such cases, the identifiers must be declared with the BUILTIN attribute when the built-in function is to be used within a preprocessor procedure.

The preprocessor built-in functions COUNTER and COMPILETIME do not require arguments and must not be given a null argument list.

COMPILETIME built-in function

COMPILETIME returns a character string, length 18, in the format of DDbMMMbYYmHH.MM.SS. The character string contains the date and the time of compilation. The syntax for COMPILETIME is:

The returned character string represents:

b	blank
DD	day of the month
MMM	month in the form JAN, FEB, MAR, etc.
YY	year
HH	hour
MM	minute
SS	second

A leading zero in the day of the month field is replaced by a blank; no other leading zeros are suppressed.

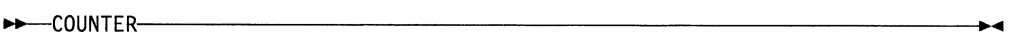
If no timing facility is available, the last 8 characters of the returned string are set to 00.00.00.

The following example shows how to print the string returned by COMPILETIME when your program is executed:

```
%DECLARE COMP_TIME CHAR;
%COMP_TIME='''||COMPILETIME||'''';
PUT EDIT (COMP_TIME) (A);
```

COUNTER built-in function

COUNTER returns a character string, length 5, containing a decimal number. The returned number is 00001 for the first invocation, and is incremented by one on each successive invocation. The syntax for COUNTER is:

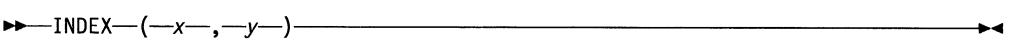


If COUNTER is invoked more than 99999 times, a diagnostic message is issued and 00000 is returned. The next invocation is treated as the first.

The COUNTER built-in function can be used to generate unique names, or for counting purposes.

INDEX built-in function

INDEX returns a FIXED value indicating the starting position within the character expression x of a substring identical to character expression y. The syntax for INDEX is:



x character expression to be searched

y character expression to be searched for.

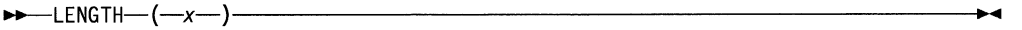
If y does not occur in x, or if either string is null, the value 0 is returned.

If y occurs more than once in x, the starting position of the leftmost occurrence is returned.

The arguments of INDEX are converted to character, if necessary.

LENGTH built-in function

LENGTH returns a FIXED value specifying the current length of a given character expression x. The syntax for LENGTH is:



x character expression. x is converted to character, if necessary.

PARMSET built-in function

The PARMSET built-in function can be used only within a preprocessor procedure. It is used to determine whether a specified parameter is set on invocation of the procedure. The syntax for PARMSET is:

x must be a parameter of the preprocessor procedure.

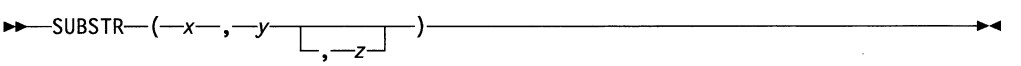
PARMSET returns a bit value of '1'B if the parameter x was explicitly set by the function reference which invoked the procedure, and a bit value of '0'B if it was not—that is, if the corresponding argument was omitted from the function reference in a preprocessor expression, or was the null string in a function reference from input text.

PARMSET can return '0'B, even if a matching argument does appear in the reference, but the reference is in another preprocessor procedure, as follows:

- If the argument is not itself a parameter of the invoking procedure, PARMSET returns the value '1'B.
- If the argument is a parameter of the invoking procedure, PARMSET returns the value for the specified parameter when the invoking procedure was itself invoked.

SUBSTR built-in function

SUBSTR returns a substring of the character expression x. The syntax for SUBSTR is:



x character expression from which the substring is extracted. x converts to character, if necessary.

y expression that can be converted to FIXED, specifying the starting position of the substring in x.

z expression that can be converted to FIXED, specifying the length of the substring in x. If z is 0, a null string is returned. If z is omitted, the substring returned is position y in x to the end of x.

If z is negative, or if the values of y and z are such that the substring does not lie entirely within x, the result is undefined.

Preprocessor statements

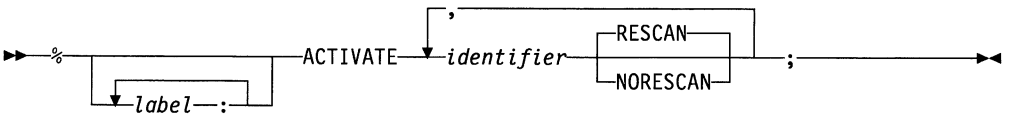
This section lists alphabetically the preprocessor statements and discusses each.

Comments can appear within preprocessor statements wherever blanks can appear. Such comments are not inserted into preprocessor output text.

All preprocessor statements can be labeled.

%ACTIVATE statement

A %ACTIVATE statement makes an identifier active and eligible for replacement. Any subsequent encounter of that identifier in the input text while the identifier is active will initiate replacement activity. The syntax for the %ACTIVATE statement is:



Abbreviation: %ACT

identifier
specifies the name of a preprocessor variable, a preprocessor procedure, or a preprocessor built-in function.

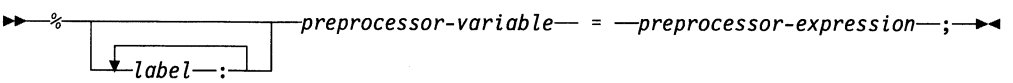
RESCAN
specifies that when the identifier is scanned by the preprocessor, replacement (as described below for NORESCAN) and rescanning takes place. RESCAN is the default.

NORESCAN
specifies that when the identifier is encountered by the preprocessor scan, it is replaced in the preprocessor output by that text which is either the current value of the variable whose name matches the identifier, or the result of invoking the function whose name matches the identifier. This text is not rescanned for further replacement.

The execution of a %ACTIVATE statement for an identifier that is already activated has no effect, except to change from RESCAN to NORESCAN or vice versa.

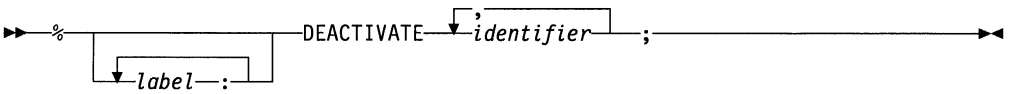
%assignment statement

A %assignment statement evaluates a preprocessor expression and assigns the result to a preprocessor variable. The syntax for the %assignment statement is:



%DEACTIVATE statement

A %DEACTIVATE statement makes an identifier inactive. The syntax for the %DEACTIVATE statement is:



Abbreviation: %DEACT

identifier
specifies the name of either a preprocessor variable, a preprocessor procedure, or a preprocessor built-in function.

The deactivation of an identifier causes loss of its replacement capability but not its value. Hence, the reactivation of such an identifier need not be accompanied by the assignment of a replacement value.

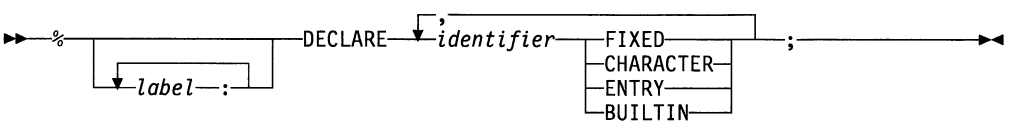
The deactivation of an identifier does not prevent it from receiving new values in subsequent preprocessor statements.

Deactivation of a deactivated identifier has no effect.

%DECLARE statement

The %DECLARE statement establishes an identifier as a preprocessor name, specifies attributes of the name, and establishes the scope of the name.

A %DECLARE statement behaves as a %ACTIVATE statement when it is encountered outside a preprocessor procedure, and activates, with the RESCAN option, all identifiers declared in the %DECLARE statement. The syntax for the %DECLARE statement is:



Abbreviations: %DCL for %DECLARE
CHAR for CHARACTER

identifier
specifies the name of either a preprocessor variable, a preprocessor procedure, or a preprocessor built-in function.

CHARACTER
specifies that the identifier represents a varying-length character string that has no maximum length.

FIXED A preprocessor variable declared with the attribute FIXED is also given the attributes DECIMAL(5,0).

ENTRY An entry declaration can be specified for each preprocessor entry name in the source program. The declaration activates the entry name.

The declaration of a preprocessor procedure entry name can be performed explicitly by its appearance as the label of a %PROCEDURE statement. This explicit declaration, however, does not activate the preprocessor procedure name.

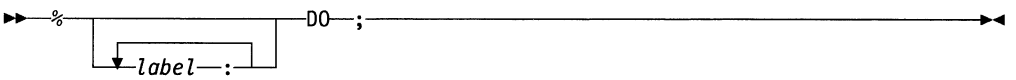
BUILTIN
specifies that the identifier is the preprocessor built-in function of the same name.

Factoring of attributes is allowed as described for DECLARE statements under "DECLARE statement" on page 152.

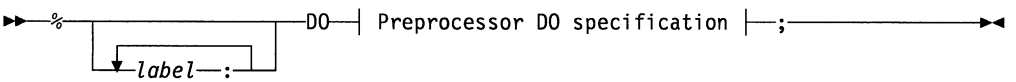
%DO statement

The %DO statement, and its corresponding %END statement, delimit a preprocessor do-group, and can also specify repetitive execution of the do-group. The syntax for the %DO statement is:

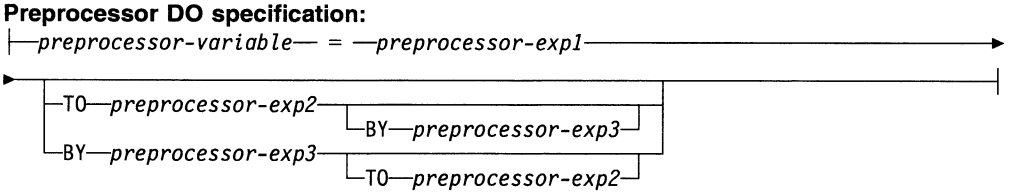
Type 1:



Type 3:



Preprocessor DO specification:



preprocessor-variable
preprocessor-exp1
preprocessor-exp2
preprocessor-exp3
have the same meaning as the corresponding variable and expressions in a DO statement (as described under "DO statement" on page 175).

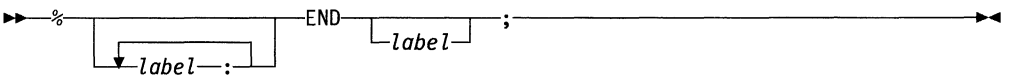
Preprocessor do-groups can be nested.

Control cannot transfer to a Type 3 preprocessor do-group, except by return from a preprocessor procedure invoked from within the do-group.

Preprocessor statements, input text, and listing control statements can appear within a preprocessor do-group. The preprocessor statements are executed; input text is scanned for possible replacement activity.

%END statement

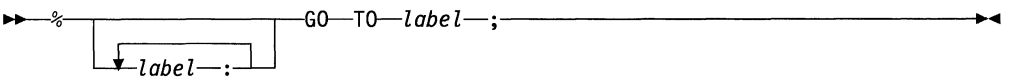
The %END statement is used in conjunction with %DO or %PROCEDURE statements to delimit preprocessor do-groups or preprocessor procedures. The syntax for the %END statement is:



The label following END must be a label of a %PROCEDURE or %DO statement. Multiple closure is allowed.

%GO TO statement

The %GO TO statement causes the preprocessor to continue its scan at the specified label. The syntax for the %GO TO statement is:



Abbreviation: %GOTO

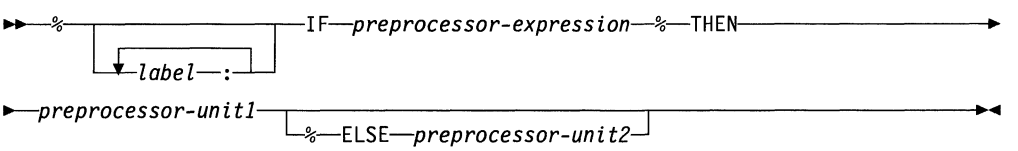
The label following the GO TO specifies the point to which the scan is transferred. It must be a label of a preprocessor statement, although it cannot be the label of a preprocessor procedure.

A preprocessor GO TO statement appearing within a preprocessor procedure cannot transfer control to a point outside of that procedure. In other words, the label following GO TO must be contained within the procedure.

See "%INCLUDE Statement" below, for a restriction regarding the use of %GO TO with included strings.

%IF statement

The %IF statement controls the flow of the scan according to the bit value of a preprocessor expression. The syntax for the %IF statement is:



preprocessor-expression
is evaluated and converted to a bit string (if the conversion cannot be made, it is an error).

preprocessor-unit
is any single preprocessor statement (other than %DECLARE, %PROCEDURE, %END, or %DO) or a preprocessor do-group. Otherwise, the description is the same as that given under "IF statement" on page 188.

If any bit in the string has the value '1'B, unit1 is executed and unit2, if present, is ignored; if all bits are '0'B, unit1 is ignored and unit2, if present, is executed.

Scanning resumes immediately following the %IF statement, unless, of course, a %GO TO or preprocessor RETURN statement in one of the units causes the scan to resume elsewhere.

%IF statements can be nested in the same manner used for nesting IF statements, as described under "IF statement" on page 188.

%INCLUDE statement

The external text specified by a %INCLUDE statement is included into the preprocessor input at the point at which the %INCLUDE statement is executed. Such text, once included, is called *included* text and can consist of preprocessor statements, listing control statements, and PL/I source.

The syntax for the %INCLUDE statement is described under "%INCLUDE statement" on page 189.

Each *type* and *member* pair identifies the external text to be incorporated into the source program.

The scan continues with the first character in the included text. The included text is scanned in the same manner as the preprocessor input. Hence, included text can contribute to the preprocessor output being formed.

%INCLUDE statements can be nested. In other words, included text can contain %INCLUDE statements.

A %GO TO statement in included text may transfer control only to a point within the same include file. The target label in the %GOTO statement must not precede the %GOTO.

Preprocessor statements, do-groups, and procedures in included text must be complete. For example, it is not allowable to have half of a %IF statement in an included text and half in another portion of the preprocessor input.

If the preprocessor input and the included text contain no preprocessor statements other than %INCLUDE, execution of the preprocessor can be omitted. (This necessitates the use of the INCLUDE compiler option. See the *PL/I VSE Programming Guide.)*

For example, assume that PAYRL is a member of a sublibrary and contains the following text (a structure declaration):

```
DECLARE 1 PAYROLL,
          2 NAME,
            3 LAST   CHARACTER (30) VARYING,
            3 FIRST  CHARACTER (15) VARYING,
            3 MIDDLE CHARACTER (3) VARYING,
          2 CURR,
            3 (REGLAR, OVERTIME) FIXED DECIMAL (8,2),
          2 YTD LIKE CURR;
```

Then the following preprocessor statements:

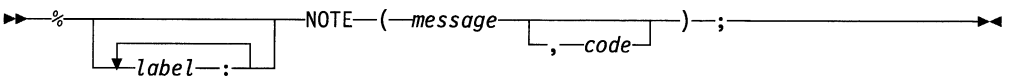
```
%DECLARE PAYROLL CHARACTER;
%PAYROLL='CUM_PAY';
%INCLUDE PAYRL;
%DEACTIVATE PAYROLL;
%INCLUDE PAYRL;
```

generate two structure declarations in the preprocessor output text. The only difference between them is their names, CUM_PAY and PAYROLL.

Execution of the first %INCLUDE statement incorporates the text in PAYRL into the preprocessor input. When the preprocessor scan encounters the identifier PAYROLL in this included text, it replaces it with the current value of the active preprocessor variable PAYROLL, namely, CUM_PAY. Further scanning of the included text results in no additional replacements. The preprocessor scan then encounters the %DEACTIVATE statement and deactivates the preprocessor variable PAYROLL. When the second %INCLUDE statement is executed, the text in PAYRL once again is incorporated into the preprocessor input. This time, however, scanning of the included text results in no replacements whatsoever.

%NOTE statement

The %NOTE statement generates a preprocessor 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 VSE Programming Guide).*

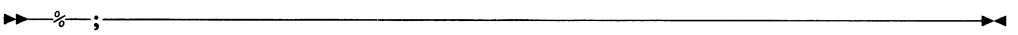
Generated messages of severity U cause immediate termination of preprocessing and compilation. Generated messages of severity S, E, or W might cause

termination of compilation, depending upon the setting of the NOSYNTAX, NOCOMPILE, and NORUN compiler options.

DBCS messages can be generated by using mixed data when the GRAPHIC compiler option is in effect.

%null statement

The %null statement does nothing and does not modify sequential statement execution. The syntax for the %null statement is:



Note: The %PROCEDURE and RETURN statements are described earlier in this chapter.

Preprocessor examples

Example 1

If the preprocessor input contains:

```
%DECLARE A CHARACTER, B FIXED;
%A = 'B+C';
%B = 2;
X = A;
```

the following is inserted into the preprocessor output:

```
X =        2+C;
```

The preprocessor statements activate A and B with the default RESCAN, assign the character string 'B+C' to A, and assign the constant 2 to B.

The fourth line is input text. The current value of A, which is 'B+C', will replace A in the preprocessor output. But this string contains the preprocessor variable B. Upon rescanning B, the preprocessor finds that it has been activated. Hence, the value 2 replaces B in the preprocessor output. The preprocessor variable B has a default precision of (5,0) and, therefore, actually contains 2 preceded by four zeros. When this value replaces B in the string 'B+C' it is converted to a character string and becomes 2 preceded by seven blanks.

Further rescanning shows that 2 cannot be replaced; scanning resumes with +C which, again, cannot be replaced.

If, in the above example, the preprocessor variable A was activated by this statement:

```
%ACTIVATE A NORESCAN;
```

the preprocessor output would be:

```
X = B+C;
```

Example 2

If the preprocessor input contains:

```
%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 preprocessor output would be as follows (replacement blanks are not shown):

```
S = I*A(I)*3;
R = 20*T*2;
```

Example 3

This example illustrates how preprocessor facilities can be used to speed up the execution of a do-group, such as:

```
DO I=1 TO 10;
Z(I)=X(I)+Y(I);
END;
```

The following would accomplish the same thing, but without the requirements of incrementing and testing during execution of the compiled program:

```
%DECLARE I FIXED;
%DO I = 1 TO 10;
Z(I)=X(I)+Y(I);
%END;
%DEACTIVATE I;
```

The third line is input text and is scanned for replacement activity. The first time that this line is scanned, I has the value 1 and has been activated. Therefore, the following is inserted into the preprocessor output:

```
Z(       1)=X(       1)+Y(       1);
```

Each 1 is preceded by seven blanks.

For each increment of I, up to and including 10, the input text is scanned and each occurrence of I is replaced by its current value. As a result, the following is inserted into the preprocessor output:

```
Z(       1)=X(       1)+Y(       1);
Z(       2)=X(       2)+Y(       2);
          .
          .
          .
Z(      10)=X(      10)+Y(      10);
```

When the value of I reaches 11, control falls through to the %DEACTIVATE statement.

Example 4

In the preprocessor input below, VALUE is a preprocessor function procedure that returns a character string of the form 'arg1(arg2)', where *arg1* and *arg2* represent the arguments that are passed to the function:

```
DECLARE (Z(10), Q) FIXED;
%A='Z';
%ACTIVATE A, VALUE;
Q = 6 + VALUE(A,3);
%DECLARE A CHARACTER;
%VALUE: PROC(ARG1,ARG2) RETURNS(CHAR);
        DCL ARG1 CHAR, ARG2 FIXED;
        RETURN(ARG1||'('||ARG2||')');
        %END VALUE;
```

When the scan encounters the fourth line, A is active and is thus eligible for replacement. Since VALUE is also active, the reference to it in the fourth line invokes the preprocessor function procedure of that name.

However, before the arguments A and 3 are passed to VALUE, A is replaced by its value Z (assigned to A in a previous assignment statement), and 3 is converted to fixed-point to conform to the attribute of its corresponding parameter. VALUE then performs a concatenation of these arguments and the parentheses and returns the concatenated value, that is, the string Z (3), to the point of invocation. The returned value replaces the function reference and the result is inserted into the preprocessor output. Thus, the preprocessor output generated is:

```
DECLARE (Z(10),Q) FIXED;
Q = 6+Z(       3);
```

Example 5

The preprocessor function procedure GEN defined below can generate a GENERIC declaration for up to 99 entry names with up to 99 parameter descriptors in the parameter descriptor lists. Only four are generated in this example.

```
%DCL GEN ENTRY;
DCL A GEN (A,2,5,FIXED);
  %GEN: PROC(NAME,LOW,HIGH,ATTR) RETURNS (CHAR);
DCL (NAME, SUFFIX, ATTR, STRING) CHAR, (LOW, HIGH, I, J) FIXED;
STRING='GENERIC(';
DO I=LOW TO HIGH;                       /* ENTRY NAME LOOP*/
   IF I>9 THEN
      SUFFIX=SUBSTR(I, 7, 2);
                                        /* 2 DIGIT SUFFIX*/
   ELSE SUFFIX=SUBSTR(I, 8, 1);
                                        /* 1 DIGIT SUFFIX*/
   STRING=STRING||NAME||SUFFIX||' WHEN (';
   DO J=1 TO I;                         /* DESCRIPTOR LIST*/
      STRING=STRING||ATTR;
      IF J<I                            /* ATTRIBUTE SEPARATOR*/
         THEN STRING=STRING||',';
         ELSE STRING=STRING||')';
                                        /* LIST SEPARATOR */
   END;
   IF I<HIGH THEN                       /* ENTRY NAME SEPARATOR*/
      STRING=STRING||',';
   ELSE STRING=STRING||')';
                                       /* END OF LIST /*
END;
RETURN (STRING)
% END;
```

The preprocessor output produced is:

```
DCL A GENERIC(A2 WHEN (FIXED,FIXED),
              A3 WHEN (FIXED, FIXED, FIXED),
              A4 WHEN (FIXED, FIXED, FIXED, FIXED),
              A5 WHEN (FIXED, FIXED, FIXED, FIXED, FIXED));
```

Example 6

This example shows a preprocessor procedure that implements a statement of the form:

```
SEARCH TABLE(array) FOR(value)
USING(variable) AND(variable);
```

This statement searches a specified two-dimensional array for a specified value, using specified or default variables for the array subscripts. After execution of the statement, the array subscript variables identify an element that contains the specified value. If no element contains the specified value, both subscript variables are set to -22222.

The preprocessor procedure that implements this statement is:

```
%SEARCH:
PROC(TABLE,FOR,USING,AND) STATEMENT RETURNS(CHARACTER);

  DECLARE(TABLE,FOR,USING,AND,LABL, DO1,DO2) CHARACTER,
      (PARMSET,COUNTER) BUILTIN;

  IF PARMSET(TABLE) & PARMSET(FOR) THEN;
  ELSE SERR:DO;
  NOTE ('MISSING OR INVALID ARGUMENT(S)'||'FOR ''SEARCH''',4);
  RETURN ('/*INVALID SEARCH STATEMENT*/');
  END;

  IF ¬PARMSET(USING) THEN
    USING='I';
  IF ¬PARMSET(AND)   THEN
    AND='J';
  IF USING = AND  THEN
    GO TO SERR;

  LABL='SL'||COUNTER;
  DO1=LABL||': DO '||USING||'=LBOUND('||TABLE||',1)
    TO HBOUND('||TABLE||',1);';
  DO2='DO '||AND||'=LBOUND('||TABLE||',2)
    TO HBOUND ('||TABLE||',2);';

  RETURN(DO1||DO2||'SELECT('||TABLE
    ||'('||USING||','||AND||'));
  WHEN('||FOR||') LEAVE '||LABL||';
  OTHER;
  END '||LABL||';
  IF '||AND||' > HBOUND('||TABLE||',2) THEN
    '||USING||','||AND||.' = -22222;');
%END SEARCH;
```

The PARMSET built-in function is used to determine which parameters are set when the procedure is invoked. If USING is not set, the default array subscript variable I is used. If AND is not set, J is used. If TABLE or FOR is not set, or if the invocation results in the same variable being used for both subscripts, a preprocessor diagnostic message is issued and a comment is returned in the preprocessor output.

The COUNTER built-in function is used to generate unique labels for the preprocessor output returned by the procedure.

The procedure can be invoked with keyword arguments or positional arguments, or a combination of the two. The following invocations of the procedure produce identical results:

```
SEARCH TABLE(LIST.NAME) FOR('J.DOE') USING(I) AND(J);

SEARCH TABLE(LIST.NAME) FOR('J.DOE');

SEARCH(LIST.NAME) FOR('J.DOE');

SEARCH(LIST.NAME,'J.DOE');

SEARCH(,'J.DOE') TABLE(LIST.NAME);
```

The preprocessor output returned by any of these invocations is:

```
SL00001:
DO I=LBOUND(LIST.NAME,1) TO HBOUND(LIST.NAME,1);
   DO J=LBOUND(LIST.NAME,2) TO HBOUND(LIST.NAME,2);
      SELECT(LIST.NAME(I,J));
   WHEN('J.DOE') LEAVE SL00001;
   OTHER;
END SL00001;
IF J > HBOUND(LIST.NAME,2) THEN
   I,J = -22222;
```

The label SL00001 is returned only for the first invocation. A new unique label is returned for each subsequent invocation.

Appendix. PL/I limits

The table below summarizes the implementation limits for the PL/I VSE language elements:

Table 36 (Page 1 of 3). PL/I language element limits

Language Element	Description	Limit
Data Aggregates (Arrays, Structures and AREAs)	Maximum size of a data aggregate	2147483648 bytes
	Maximum size of a data aggregate which contains UNALIGNED BIT data	268435455 bytes
	Maximum number of dimensions for an array	15
	Minimum lower bound for an array	-2147483648
	Maximum upper bound for an array	+2147483647
	Maximum number of levels in a structure	15
	Maximum level number in a structure	255
String Data	Maximum length of a CHARACTER variable	32767 characters
	Maximum length of a BIT variable	32767 bits
	Maximum length of a GRAPHIC variable	16383 graphics
	Maximum string repetition factor	32767
	The maximum number of bytes in the external representation of any string constant is 3600. This corresponds to the following limits for each string constant: BIT BX CHARACTER X GRAPHIC GX M The external representation includes all quotes and string suffixes, as well as DBCS shift codes. 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.	 450 bits 14400 bits 3600 characters 1800 characters 1800 graphics 900 graphics 3600 bytes
	The maximum number of bytes in the external representation of any preprocessor string constant is 4096. This corresponds to the following limits for each string constant: CHARACTER X GRAPHIC	 4096 characters 2048 characters 2048 graphics
	Maximum number of picture characters in a character PICTURE	511
	Maximum number of picture characters in a numeric PICTURE	256
	Maximum number of numeric picture characters in a numeric PICTURE	15

Table 36 (Page 2 of 3). PL/I language element limits

Language Element	Description	Limit
Arithmetic Precisions	Maximum precision for FIXED DECIMAL	15
	Maximum precision for FIXED BINARY	31
	Maximum precision for FLOAT DECIMAL	33
	Maximum precision for FLOAT BINARY	109
	Maximum scale factor for FIXED data	127
	Minimum scale factor for FIXED data	-128
Program Size	Maximum length of an identifier	31 bytes
	Maximum length of an external name While external names generated by PL/I are always 8 characters in length, user-defined external names cannot exceed 7 characters. This prevents duplicate definition of names. If a name of more than 7 characters is declared with the EXTERNAL attribute, the first 4 characters are concatenated with the last 3 characters to form the EXTERNAL name. However, an entry name declared with ASSEMBLER or COBOL specified in the OPTIONS attribute can have up to 8 characters. If more than 8 characters are specified, only the leftmost 8 are used.	7 bytes
	Maximum number of procedures in a program	255
	Maximum number of statements in a program	10,000
	Maximum number of DEFAULT statements in a block	31
	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	25
	Maximum number of arguments in a CALL statement or function reference	64
	Maximum number of parameters for a procedure	64
	Maximum number of parameters for a preprocessor procedure	63
	Maximum nesting of factored attributes	15
	Maximum nesting of BEGIN and PROCEDURE blocks	42
	Maximum nesting of DO groups	38
	Maximum nesting of IF statements	80
	Maximum nesting of SELECT statements	50
	Maximum nesting of %INCLUDE statements	8

Table 36 (Page 3 of 3). PL/I language element limits

Language Element	Description	Limit
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 and PRECISION functions	same as corresponding limit for arithmetic precision
	Maximum values for the scale (q) in the ADD, BINARY, DECIMAL, DIVIDE, FIXED, MULTIPLY and PRECISION 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
Miscellaneous	Maximum length for a KEYTO character string	120
	Maximum length for a KEYTO graphic string	60
	Maximum line size for LINESIZE option	32000
	Minimum line size for LINESIZE option	10
	Maximum page size for PAGESIZE option	32000
	Minimum page size for PAGESIZE option	1
	Maximum size of DISPLAY character string	126 bytes
	Maximum length for a DISPLAY REPLY message	72 bytes
	Maximum length for a %NOTE message	256 bytes

Bibliography

IBM PL/I for VSE/ESA publications

Fact Sheet, GC26-8052

Installation and Customization Guide, SC26-8057

Licensed Program Specifications, GC26-8055

Language Reference, SC26-8054

Compile-Time Messages and Codes, SC26-8059

Diagnosis Guide, SC26-8058

Migration Guide, SC26-8056

Programming Guide, SC26-8053

Reference Summary, SX26-3836

IBM Language Environment for VSE/ESA publications

Concepts Guide, GC26-8063

Fact Sheet, GC26-8062

Debugging Guide and Run-Time Messages, SC26-8066

Diagnosis Guide, SC26-8060

Installation and Customization Guide, SC26-8064

Licensed Program Specifications, GC26-8061

Programming Guide, SC26-8065

Reference Summary, SX26-3835

VSE/ESA publications

VSE/ESA Version 1

Administration, SC33-6505

Messages and Codes, SC33-6507

System Control Statements, SC33-6513

System Utilities, SC33-6517

System Macros Reference, SC33-6516

Guide to System Functions, SC33-6511

VSE/VSAM Commands and Macros, SC33-6532

VSE/VSAM User's Guide, SC33-6535

VSE/ESA Version 2

Administration, SC33-6605

Messages and Codes, SC33-6607

System Control Statements, SC33-6613

System Utilities, SC33-6617

System Macros Reference, SC33-6616

Guide to System Functions, SC33-6611

VSE/VSAM Commands and Macros, SC33-6631

VSE/VSAM User's Guide, SC33-6632

Related publications

CICS/VSE

Application Programming Guide, SC33-0712

Application Programming Reference, SC33-0713

System Definition and Operations Guide, SC33-0706

Resource Definition, SC33-0708

DFSORT for VSE/ESA

Application Programming Guide, SC26-7040

Sort/Merge II

DOS/VS VM/SP Sort/Merge Version 2 Application Programming Guide, SC33-4044

DL/I DOS/VS

Application Programming: High Level Programming Interface, SH24-5009

Application Programming: CALL and RQDLI Interfaces, SH12-5411

SQL/DS

SQL/Data System Application Programming Guide for VSE, SH09-8098

Softcopy publications

These collections contain the LE/VSE and LE/VSE-conforming language product publications:

VSE Collection, SK2T-0060

Application Development Collection, SK2T-1237

Glossary

This glossary defines terms for all platforms and releases of PL/I. It might contain terms that this manual does not use. If you do not find the terms for which you are looking, see the index in this manual or the *IBM Dictionary of Computing*, SC20-1699.

References: This glossary uses the following references:

Contrast with: Indicates a term or terms that have an opposed or substantially different meaning.

See: Refers to a multiple-word term in which this term appears.

See also: Refers to related terms that have similar (but not synonymous) meanings.

A

access. To reference or retrieve 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 the appropriate condition is raised.

activate (a block). To initiate the execution of a block. A procedure block is activated when it is invoked. A begin block is activated when it is encountered in the normal flow of control, including a branch.

activate (a preprocessor variable or preprocessor entry point). To make a macro facility identifier eligible for replacement in subsequent source code. The %ACTIVATE statement activates preprocessor variables or preprocessor entry points.

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. (3) The state in which an event variable is said to be during the time it is associated with an asynchronous operation.

actual origin (AO). The location of the first item in the array or structure.

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. The bound (of an array), the length (of a string), or the 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 main storage is associated and not freed.

allocation. (1) The reservation of main storage for a variable. (2) A generation of an allocated variable. (3) The association of a PL/I file with a system data set, device, or file.

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 portion of storage within which based variables can be allocated.

argument. An expression in an argument list as part of an invocation of a subroutine or function.

argument list. A parenthesized list of zero 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 becomes the parameter list of the entry point.

arithmetic comparison. A comparison of 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: + – * / **

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.

assignment. The process of giving a value to a variable.

asynchronous operation. (1) The overlap of an input/output operation with the execution of statements. (2) 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 a task to be interrupted.

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. A member of a structure or a union that is itself not another structure or union.

base item. The automatic, controlled, or static variable or the parameter upon which a defined variable is defined.

based reference. A reference that has the based storage class.

based storage allocation. The allocation of storage for based variables.

based variable. A variable whose storage address is provided by a locator. 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, forming a name scope. A begin-block is activated either by the raising of a condition (if the begin-block is the action specification for an ON-unit) 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 digit. See *bit*.

binary fixed-point value. An integer consisting of binary digits and having an optional binary point and optional sign. 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 0 or a 1. (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 string constant. (1) A series of binary digits enclosed in single quotes and followed immediately by the suffix B. Contrast with *character constant*. (2) A series of hexadecimal digits enclosed in single quotes and followed by the suffix B4 or BX.

bit string. A string composed of zero or more bits.

bit string operators. The logical operators not (¬), and (&), and or (|).

bit value. A value that represents a bit type.

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. A block can be a procedure or a begin-block.

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 SQRT (square root).

built-in function reference. A built-in function name, which has an optional argument list.

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.

C

call. To invoke a subroutine by using the CALL statement or CALL option.

character comparison. A left-to-right, character-by-character comparison according to the collating sequence. See also *arithmetic comparison, bit comparison*.

character string constant. A sequence of characters enclosed in single quotes; for example, 'Shakespeare''s "Hamlet"'.

character set. A defined collection of characters. See *language character set* and *data character set*. See also *ASCII* and *EBCDIC*.

character string picture data. Picture data that has only a character value. This type of picture data must have at least one A or X picture specification character. Contrast with *numeric picture data*.

closing (of a file). The dissociation of a file from a data set or device.

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.

combined nesting depth. The deepest level of nesting, determined by counting the levels of 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 locator, 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, 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 object module generated, the types of printed output produced, and so forth.

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, ON, OTHERWISE, and WHEN 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 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 or disabled.

connected aggregate. An array or structure whose elements occupy contiguous storage without any intervening data items. Contrast with *nonconnected aggregate*.

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 identifier declared with the FILE or the ENTRY attribute but without the VARIABLE attribute.

constant reference. A value reference which has a constant as its object.

contained block, declaration, or source text. All blocks, procedures, statements, declarations, or source text inside a procedure or a package block. The entire procedure, and the BEGIN statement and its corresponding END statements are not contained in the block.

containing block. The procedure or begin block that contains the declaration, statement, procedure, or other source text in question.

contextual declaration. The appearance of an identifier that has not been explicitly declared in a DECLARE statement, but whose context of use allows the association of specific attributes with the identifier.

control character. A character in a character set whose occurrence in a particular context specifies a control function. One example is the end-of-file (EOF) marker.

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 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. For example, converting a character string to an arithmetic value such as FIXED BINARY (15,0).

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. The generation of an automatic or controlled variable that is currently available by referring to the name of the variable.

D

data. Representation of information or of value in a form suitable for processing.

data aggregate. A data item that is a collection of other data items.

data attribute. A keyword that specifies the type of data that the data item represents, such as FIXED BINARY.

data-directed transmission. The type of stream-oriented transmission in which data is transmitted. It resembles an assignment statement and is of the form:

```
name = constant
```

data item. A single named unit of data.

data list. In stream-oriented transmission, a parenthesized list of the data items used in GET and PUT statements. Contrast with *format list*.

data set. (1) A collection of data external to the program that can be accessed by reference to a single file name. (2) A device that can be referenced.

data specification. The portion of a stream-oriented 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.

DBCS. In the character set, each character is represented by two consecutive bytes.

deactivated. The state in which an identifier is said to be when its value cannot replace a preprocessor identifier in source program text. Contrast with *active*.

debugging. Process of removing bugs from a program.

decimal. The number system whose numerals are 0 through 9.

decimal digit. One of the digits 0 through 9.

decimal digit picture 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. See *numeric picture data*.

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 variable. A variable that is associated with some or all of the storage of 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, blank, pointer, asterisk, and single quote. They define the limits of identifiers, constants, picture specifications, iSUBs, and keywords.

descriptor. A control block that holds information about a variable, such as area size, array bounds, or string length.

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.

disabled. The state of a condition in which no interrupt occurs and no established action will take place.

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 the condition can cause an interrupt and then invocation of the appropriate established ON-unit.

enclave. In Language Environment, an independent collection of routines, one of which is designated as the main routine. An enclave is roughly analogous to a program or run unit.

entry constant. (1) The label prefix of a PROCEDURE statement (an entry name). (2) The declaration of a name with the ENTRY attribute but without the VARIABLE attribute.

entry data. 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) An identifier that 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. See *primary entry point* and *secondary entry point*.

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 or variable; 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 in containing blocks.

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.

event. An activity in a program whose status and completion can be determined from an associated event variable.

event variable. A variable with the EVENT attribute that may be associated with an event. Its value indicates whether the action has been completed and the status of the completion.

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. The 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 size of the target area if this area were to be assigned to a target 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.

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 with the FILE attribute but not the VARIABLE attribute.

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 value of the type file.

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.

format. 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 constant. The label prefix on a FORMAT statement.

format data. A variable with the FORMAT attribute.

format label. The label prefix on a FORMAT statement.

format list. In stream-oriented transmission, a list specifying the format of the data item on the external medium. Contrast with *data list*.

fully-qualified name. A name that includes all the names in the hierarchical sequence above the member to which the name refers, as well as the name of the member itself.

function (procedure). (1) A procedure that has a RETURNS option in the PROCEDURE statement. (2) A name declared with the RETURNS attribute. 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

hex. See *hexadecimal digit*.

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.

hexadecimal digit. One of the digits 0 through 9 and A through F. A through F represent the decimal values 10 through 15, respectively.

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 26 alphabetic characters and extralingual characters, if any. The other characters, if any, can additionally include extended alphabetic, digit, or the break character.

IEEE. Institute of Electrical and Electronics Engineers.

implicit. The action taken in the absence of an explicit specification.

implicit action. The action taken when an enabled condition is raised and no ON-unit is currently established for the condition. Contrast with *ON-statement action*.

implicit declaration. A name not explicitly declared in a DECLARE statement or contextually declared.

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, union, or element, those dimensions that are derived from the containing structures. If the name is an element that is not an array, the dimensions consist entirely of its inherited dimensions. If the name is an element that is an array, its dimensions consist of its inherited dimensions plus its explicitly declared dimensions. A structure with one or more inherited dimensions is called a nonconnected aggregate. Contrast with *connected aggregate*.

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, the insertion character is used for checking purposes.

integer. (1) An optionally-signed sequence of digits or a sequence of bits without a decimal or binary point. (2) An optionally-signed whole number, commonly described as FIXED BINARY (p,0) or FIXED DECIMAL (p,0).

integral boundary. A byte multiple address of any 8-bit unit on which data can be aligned. It usually is a half-word, full-word, or double-word (2-, 4-, or 8-byte multiple respectively) boundary.

interleaved array. An array that refers to nonconnected storage.

interleaved subscripts. Subscripts that exist in levels other than the lowest level of a subscripted qualified reference.

internal block. A block that is contained in another block.

internal name. A name that is known only within the block in which it is declared, and possibly within any contained blocks.

internal procedure. A procedure that is contained in another block. Contrast with *external procedure*.

interrupt. The redirection of the program's flow of control 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 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 the given value. (2) In a format list, an expression that specifies the number of times a given format item or list of format 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 in PL/I 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. (1) A name prefixed to a statement. A name on a PROCEDURE statement is called an entry constant; a name on a FORMAT statement is called a format constant; a name on other kinds of statements is called a label constant. (2) A data item that has the LABEL attribute.

label constant. A name written as the label prefix of a statement (other than PROCEDURE, ENTRY, FORMAT, or PACKAGE) so that, during execution, program control can be transferred to that statement through a reference to its label prefix.

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

label prefix. A label prefixed to a statement.

label variable. A variable declared with the LABEL attribute. Its value is a label constant in the program.

leading zeroes. Zeros that have no significance in an arithmetic value. All zeros to the left of the first non-zero in a number.

level-number. A number that precedes a name in a DECLARE statement and specifies its relative position in the hierarchy of structure names.

level-one variable. (1) A major structure or union name. (2) Any unsubscripted variable not contained within a structure or union.

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 control block that holds the address of a variable or its descriptor.

locator/descriptor. A locator followed by a descriptor. The locator holds the address of the variable, not the address of the descriptor.

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. It might be an implicit reference.

locator value. A value that identifies or can be used to identify 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.

logical level (of a structure or union 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.

member. A structure, union, or element name, possibly dimensioned, in a structure or union.

minor structure. A structure that is contained within another structure or union. The name of a minor structure is declared with a level number greater than one and greater than its parent structure or union.

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.

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 program to execute more than one PL/I procedure simultaneously.

N

name. Any identifier that the user gives 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.

nonconnected storage. Storage occupied by nonconnected data items. For example, interleaved arrays and structures with inherited dimensions are in nonconnected storage.

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, graphic, or bit string with a length of zero.

numeric-character data. See *decimal picture data.*

numeric picture data. Picture data that has an arithmetic value as well as a character value. This type of picture data cannot contain the characters "A" or "X".

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 that is executed when the condition is raised. When the ON-statement is encountered in the flow of control for the program, it executes, establishing the action for the condition. The action executes when the condition is raised if the ON-unit is still established. Contrast with *implicit action.*

ON-unit. The specified action to be executed when the appropriate condition is raised.

opening (of a file). The association of a file with a data set.

operand. The value of an identifier, constant, or an expression to which an operator is applied, possibly in conjunction with another operand.

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, graphics, or bits concatenated to the right of a string to extend the string to a required length. (2) One or more bytes or bits inserted in a structure or union so that the following element within the structure or union is aligned on the appropriate integral boundary.

parameter. A name in the parameter list following the PROCEDURE statement, specifying an argument that will be passed when the procedure is invoked.

parameter descriptor. The set of attributes specified for a 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 or union member to which the name refers, as well as the name of the member itself.

picture data. Numeric data, character data, or a mix of both types, represented in character form.

picture specification. A data item that is described using the picture characters in a declaration with the PICTURE attribute or in a P-format item.

picture specification character. Any of the characters that can be used in a picture specification.

PL/I character set. A set of characters that has been defined to represent program elements in PL/I.

point of invocation. The point in the invoking block at which the reference to the invoked procedure appears.

pointer. A type of variable that identifies a location in storage.

pointer value. A value that identifies the pointer type.

pointer variable. A locator variable with the POINTER attribute that contains a pointer value.

precision. The number of digits or bits 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 before the compilation takes place.

preprocessor statement. A special statement appearing in the source program that specifies the actions to be performed by the preprocessor. It is executed as it is encountered by the preprocessor.

primary entry point. The entry point identified by any of the names in the label list of the PROCEDURE statement.

problem data. Coded arithmetic, bit, character, graphic, and picture data.

problem-state program. A program that operates in the problem state of the operating system. It does not contain input/output instructions or other privileged instructions.

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 the procedure or one of its entry names. See also *external procedure* and *internal procedure*.

procedure reference. An entry constant or variable. It may be followed by an argument list. It may appear in a CALL statement or the CALL option, or as a function reference.

process. The highest level of the Language Environment program management model. A process is a collection of resources, both program code and data, and consists of at least one enclave.

program. A set of one or more external procedures or packages. One of the external procedures 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.

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. It is usually on the left-hand side of an assignment statement.

Q

qualified name. A hierarchical sequence of names of structure or union members, connected by periods, used to identify a name within 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 statement apply.

record. (1) The logical unit of transmission in a record-oriented input or output operation. (2) A collection of one or more related data items. The items usually have different data attributes and usually are described by a structure or union declaration.

recorded key. A character string identifying a record in a direct-access data set where the character string itself is also recorded as part of the data.

record-oriented data transmission. The transmission of data in the form of separate records. Contrast with *stream data transmission*.

recursive procedure. A procedure that can be called from within itself or from within another active procedure.

reentrant procedure. A procedure that can be activated by multiple tasks, threads, or processes simultaneously without causing any interference between these tasks, threads, and processes.

REFER expression. The expression preceding the keyword REFER, which is used as the bound, length, or size when the based variable containing a REFER option is allocated, either by an ALLOCATE or LOCATE statement.

REFER object. The variable in a REFER option that holds or will hold the current bound, length, or size for the member. The REFER object must be a member of the same structure or union. It must not be locator-qualified or subscripted, and it must precede the member with the REFER option.

reference. The appearance of a name, except in a context that causes explicit declaration.

relative virtual origin (RVO). The actual origin of an array minus the virtual origin of an array.

remote format item. The letter R followed by the label (enclosed in parentheses) of a 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 integer constant that specifies:

1. The number of times the string constant that follows is to be repeated.
2. The number of times the picture character that follows is to be repeated.

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 that can be evaluated by the compiler during compilation, resulting in a constant. Operands of such an expression are constants, named constants, and restricted expressions.

returned value. The value returned by a function procedure.

RETURNS descriptor. A descriptor used in a RETURNS attribute, and in the RETURNS option of the PROCEDURE and ENTRY statements.

S

scalar variable. A variable that is not a structure, union, or array.

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.

scaling factor. See *scale factor*.

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 a particular name is known.

scope (of a name). See *scope (of a declaration)*.

secondary entry point. An entry point identified by any of the names in the label list of an entry statement.

select-group. A sequence of statements delimited by SELECT and END statements.

selection clause. A WHEN or OTHERWISE clause of a select-group.

self-defining data. An aggregate that contains data items whose bounds, lengths, and sizes are determined at program execution time and are stored in a member of the aggregate.

separator. See *delimiter*.

sign and currency symbol characters. The picture specification characters S, +, –, and $.

simple parameter. A parameter for which no storage class attribute is specified. It may represent an argument of any storage class, but only the current generation of a controlled argument.

simple statement. A statement other than IF, ON, WHEN, and OTHERWISE.

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 source program processors and the compiler.

source variable. A variable whose value participates in some other operation, but is not modified by the operation. Contrast with *target variable*.

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 PL/I 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 PL/I statement, composed of keywords, delimiters, identifiers, operators, and constants, and terminated by a semicolon (;). Optionally, it may have a condition prefix list and a list of labels. See also *keyword statement, assignment statement*, and *null statement*.

statement body. A statement body can be either a simple or a compound statement.

statement label. See *label constant*.

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 data is treated as though it were a continuous stream of individual data values in character form. Contrast with *record-oriented data transmission*.

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

string variable. A variable declared with the BIT, CHARACTER, or GRAPHIC attribute, whose values can be either bit, character, or graphic 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 that has the dimension attribute.

structure member. See *member*.

structuring. The hierarchy of a structure, in terms of the number of members, the order in which they appear, their attributes, and their logical level.

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 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. If the subscript is an asterisk, it specifies all of the elements of the dimension.

subscript list. A parenthesized list of one or more subscripts, one for each dimension of the array, which together uniquely identify either a single element or cross section of the array.

synchronous. A single flow of control for serial execution of a program.

T

target reference. A reference that designates a receiving variable (or a portion of a receiving variable).

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.

thread. The basic run-time path within the Language Environment program management model. It is dispatched by the system with its own instruction counter and registers. The thread is where actual code resides.

truncation. The removal of one or more digits, characters, graphics, 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

undefined. Indicates something that a user must not do. Use of a undefined feature is likely to produce different results on different implementations of a PL/I product. In that case, the application program is considered to be in error.

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 origin (VO). The location where the element of the array whose subscripts are all zero are held. If such an element does not appear in the array, the virtual origin is where it would be held.

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

We'd Like to Hear from You

IBM PL/I for VSE/ESA
Language Reference
Release 1

Publication No. SC26-8054-00

Please use one of the following ways to send us your comments about this book:

- Mail—Use the Readers' Comments form on the next page. If you are sending the form from a country other than the United States, give it to your local IBM branch office or IBM representative for mailing.
- Fax—Use the Readers' Comments form on the next page and fax it to this U.S. number: 800-426-7773.
- Electronic mail—Use one of the following network IDs:
  - Internet: COMMENTS@VNET.IBM.COM

  Be sure to include the following with your comments:
  - Title and publication number of this book
  - Your name, address, and telephone number if you would like a reply

Your comments should pertain only to the information in this book and the way the information is presented. To request additional publications, or to comment on other IBM information or the function of IBM products, please give your comments to your IBM representative or to your IBM authorized remarketer.

IBM may use or distribute your comments without obligation.

Readers' Comments

IBM PL/I for VSE/ESA
Language Reference
Release 1

Publication No. SC26-8054-00

How satisfied are you with the information in this book?

	Very Satisfied	Satisfied	Neutral	Dissatisfied	Very Dissatisfied
Technically accurate	☐	☐	☐	☐	☐
Complete	☐	☐	☐	☐	☐
Easy to find	☐	☐	☐	☐	☐
Easy to understand	☐	☐	☐	☐	☐
Well organized	☐	☐	☐	☐	☐
Applicable to your tasks	☐	☐	☐	☐	☐
Grammatically correct and consistent	☐	☐	☐	☐	☐
Graphically well designed	☐	☐	☐	☐	☐
Overall satisfaction	☐	☐	☐	☐	☐

Please tell us how we can improve this book:

May we contact you to discuss your comments? ☐ Yes ☐ No

Name

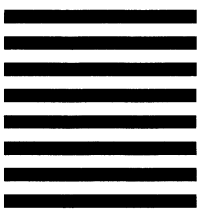
Address

Company or Organization

Phone No.

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

Department J58
International Business Machines Corporation
PO BOX 49023
SAN JOSE CA 95161-9945

Fold and Tape **Please do not staple** Fold and Tape

Readers' Comments

IBM PL/I for VSE/ESA
Language Reference
Release 1

Publication No. SC26-8054-00

How satisfied are you with the information in this book?

	Very Satisfied	Satisfied	Neutral	Dissatisfied	Very Dissatisfied
Technically accurate	☐	☐	☐	☐	☐
Complete	☐	☐	☐	☐	☐
Easy to find	☐	☐	☐	☐	☐
Easy to understand	☐	☐	☐	☐	☐
Well organized	☐	☐	☐	☐	☐
Applicable to your tasks	☐	☐	☐	☐	☐
Grammatically correct and consistent	☐	☐	☐	☐	☐
Graphically well designed	☐	☐	☐	☐	☐
Overall satisfaction	☐	☐	☐	☐	☐

Please tell us how we can improve this book:

May we contact you to discuss your comments? ☐ Yes ☐ No

Name

Address

Company or Organization

Phone No.

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

Department J58
International Business Machines Corporation
PO BOX 49023
SAN JOSE CA 95161-9945

Fold and Tape **Please do not staple** Fold and Tape

Program Number: 5686-069

Printed in U.S.A.

IBM PL/I for VSE/ESA Publications

GC26-8052	Fact Sheet
GC26-8055	Licensed Program Specifications
SC26-8056	Migration Guide
SC26-8057	Installation and Customization Guide
SC26-8053	Programming Guide
SC26-8054	Language Reference
SX26-3836	Reference Summary
SC26-8058	Diagnosis Guide
SC26-8059	Compile-Time Messages and Codes

SC26-8054-00