IBM

8100 Information System

Distributed Processing Development System (DPDS)

Programming Language for Distributed Systems (PL/DS) Reference

SC27-0446-0
File No. 8100/S370-31

SC27-0446-0
File No. 8100/S370-31

Distributed Processing Development System (DPDS)

Programming Language for Distributed Systems (PL/DS) Reference

Program Number: 5799-AZL

PRPQ Number: P88016

CONTENTS

FIGURES

INTRODUCTION

PL/DS is a System/370-resident compiler that provides flexible preparation of programs for use on the IBM 8100 Information System. It provides the opportunity for programmer productivity associated with the System/370 interactive environment. Programs can be written for use either under 8100 Distributed Program Executive (DPPX) Base, or for use under other 8100 operating environments.

PL/DS has these basic features:

- It is a high level programming language with preprocessor-like macro support.
- It enables coding at the machine level with (a) support of machine instructions and (b) support of a subset of 8100 DPPX Assembler code.
- The PL/DS object code executes on both the System/370-resident Program Development Simulator -- for testing and debugging -- and on the 8100.
- It provides simplified intra-modular linkage for programming that will be executed under DPPX Base.
- It has code optimization options.

Compiling and testing on the simulator permits an application development strategy whereby live 8100 production programs are not degraded due to new application development. This is especially important when the 8100s presently installed are running to capacity with existing production applications. New programs can be developed and tested, using System/370, before delivery of the 8100s they will run on.

The 8100 DPPX Assembler provides for coding main-line logic, invoking DPPX macros, and user definition of DPPX Assembler macros. In contrast to the DPPX Assembler, the PL/DS compiler supports:

- A high level coding language.
- IBM-supplied PL/DS macros or user-written PL/DS macros.
- 8100 Information System machine language.
- A subset of DPPX Assembler statements.

The PL/DS object code can optionally be prepared for execution on either the simulator, or on the 8100 system.

The high level compiler language provides PL/I-like coding power, and includes structured programming commands such as IF, THEN, and DO. The formatted listing option logically indents structured code for easy reading.

The PL/DS macro language permits compile time modification of PL/DS source, and supports: (a) PL/DS macro definitions stored in library data sets; (b) user definition of PL/DS macros; and (c) submittal of a subset of DPPX Assembler statements.

The machine language support has the flexibility of accepting IBM 8100 instruction mnemonics without insisting upon machine-code operands; PL/DS-like expressions are accepted as operands.

The machine language support and the DPPX Assembler statement support permit programming when exacting requirements exist, while the high level PL/DS statements meet the requirements of system level programmers. Input to the compiler is free-form, with flexible spacing allowed and the use of semicolons as statement delimiters.

The three PL/DS phases -- macro, compile, and assembly -- can optionally be individually suppressed, and there are options to obtain listings, formatted or unformatted, after the macro phase, after the compile phase, and after the assembly phase. For listing readability, there are two features:

- The formatting option for structured code, as described above.
- An option for printing included code after the program segment that included it, so that the continuity of the including segment logic is kept compact and uninterrupted.

One feature of the compiler is optimization. This option generates highly optimized code, allowing a skilled PL/DS programmer to produce very efficient code. (The user of OPTIMIZE should read "Optimization Precautions," in Appendix F, before using OPTIMIZE.)

CHAPTER 1. LANGUAGE DESCRIPTION AND CODING RULES

LANGUAGE DESCRIPTION

There are a number of sections to PL/DS language. The following descriptions will help you keep them sorted out. Refer to Figure 1 as you read this. As your familiarity with the parts of the language increases you can continue to use this figure for clarifying language relationships.

MACRO AND COMPILE CODE OBJECTIVES

The two main sections of the language are compile source code and macro source code. The compile code achieves the main objective of programming efforts: it expresses the program's logic in code that the compiler converts to a machine language object program. The macro code gives a preprocessor-like option to make changes to compile code before compiling it. Macro statements express logic to: (a) control macro phase processing, (b) to change character strings in compile source code, and (c) to add statements to compile code.

Both macro and compiler code do not have to be present in every source data set; either one can exist alone. A source data set might contain compile code only. Or, the data set might be only powerful macros that the programmer invokes and codes appropriate parameters for; these macros would produce all the compile code for the object program.

THE COMPILE CODE

The basic part of compile source code is the compile statements. Compile statements are the language that you as a programmer use to express the logic of your program.

Another part of compile code is built-in functions. These functions are compiler-provided shortcuts to conserve code and give you additional data. You code them in your compile statements.

Machine instructions are an extension of the PL/DS compile code. They enable you to explicitly code specific machine instructions, while permitting the use of conventional PL/DS expressions as operands of the instructions. The compile phase of the PL/DS compiler resolves these expressions.

DPPX Assembly statement subset support enables you to submit pure assembler code in your program, and you exercise full control in your specification of normal assembler operands. Your assembler code is assembled with no compile phase scrutiny for PL/DS expressions.

In Figure 1, the assembly statements are shown within macro definitions. This is for accuracy: the assembler statements must be inserted into the source statements by use of the PL/DS macro definitions described below.

Any of the compile code can be altered during a compiler run with PL/DS macro code.

THE MACRO CODE

Macro code is for altering compile code just before it is compiled. The macro code has three basic parts: macro definitions, macro outer statements, and macro invocations.

The logic in macro outer code is executed during the macro processing phase of the compiler. You can use macro outer statements for some macro logic, but the macro definitions are more powerful.

PL/DS Macro definitions are macro code "procedures" that you are using more than once. The operations that a macro definition contains are executed during the macro processing phase of the compiler wherever you code a macro invocation for that macro.

Within a macro definition, you use PL/DS macro definition statements to code the operations that happen when that macro is invoked. Within the macro definition statements, you can use the PL/DS macro definition functions. The macro definition functions, like the compile statement built-in functions, provide additional flexibility to the macro definition statements.

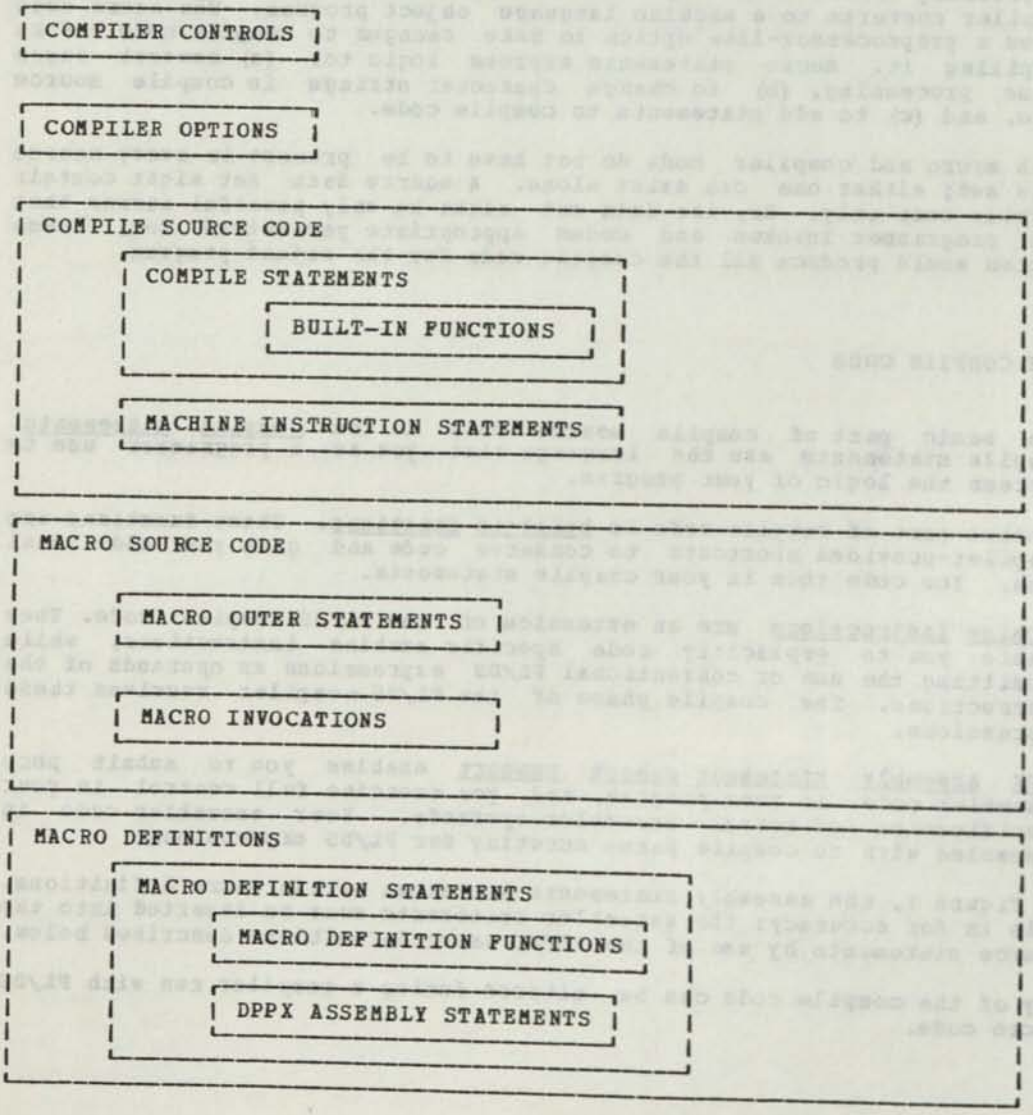


Figure 1. PL/DS Language Component Domains

PL/DS ENVIRONMENTS AND RESERVED WORDS

There are three "environments" that impact your coding. They are:

- The compile environment.
- The macro outer environment.
- The macro definition environment.

Certain rules are not shared across environments. This is most apparent in the use of keywords, reserved words, and variables. In a given environment, you can use reserved words from other environments as long as you do not violate any rules set down for their use in the present environment.

There is an important exception to this in the sharing of variables declared EXTERNAL between macro environments, and this is covered in the descriptions of the macro DECLARE and %DECLARE statements.

THE PROCESSING OF SOURCE DATA BY THE COMPILER

The effects of the compiler on the contents of the source data set are shown in Figure 2. The changes that take place in each part of the data set at each compiler phase are shown below. (The relative size of each type of code is for illustration only, and in actual cases can deviate markedly from what the figure shows.) (The actual code is not in separate groups in the input data set, but is intermixed throughout to obtain the results desired by the programmer.)

Parenthesized material points to the parts of the language that are used to obtain these results.

1. MACRO PROCESSING PHASE CHANGES:

- Machine instruction statements (see, "Machine Instruction Support") can be added from private libraries. The machine instruction is not processed, but it can be altered and additional statements inserted (macro variables and answer text).
- Compile statements (see, "Language Instructions and Built-In Functions") can be added from private libraries. The compiler code is not processed, but it can be altered and additional statements inserted.
- DPPX assembler statements can be inserted. The assembler code can be altered.
- Macro code can be added from private libraries, and/or by invocations of library-resident macro definitions. It is all executed to produce answer text and macro variable substitution values, and disappears at the end of the macro processing phase.

2. COMPILE PHASE CHANGES:

- Machine instruction statements can be added from private libraries. The compile phase processes each input machine language statement to yield the assembler statement equivalent of each actual machine language operation specified, plus, where necessary, the housekeeping assembler statements to supply the PL/DS variable values referenced in the statement. The machine language statements all become assembler statements by the end of the compile phase.
- Compile statements can be added from private libraries. The compile code is all processed into assembler statements, and disappears at the end of the compile phase.

- The addition to assembler statements is the output of the compile phase processing of both the compile source code and the machine language statements.

3. ASSEMBLY PHASE CHANGES:

- Assembler statements are converted into object code, and disappear.

SOURCE DATA SET DIFFERENCES

All of the components shown in Figure 2 do not have to be present in a source data set. One extreme could be a source data set with only compile source statements. Another extreme could be a macro-code-only source data set. (The macro invocations could generate any of: machine language, compiler code, and/or assembler code, for subsequent processing as just described.)

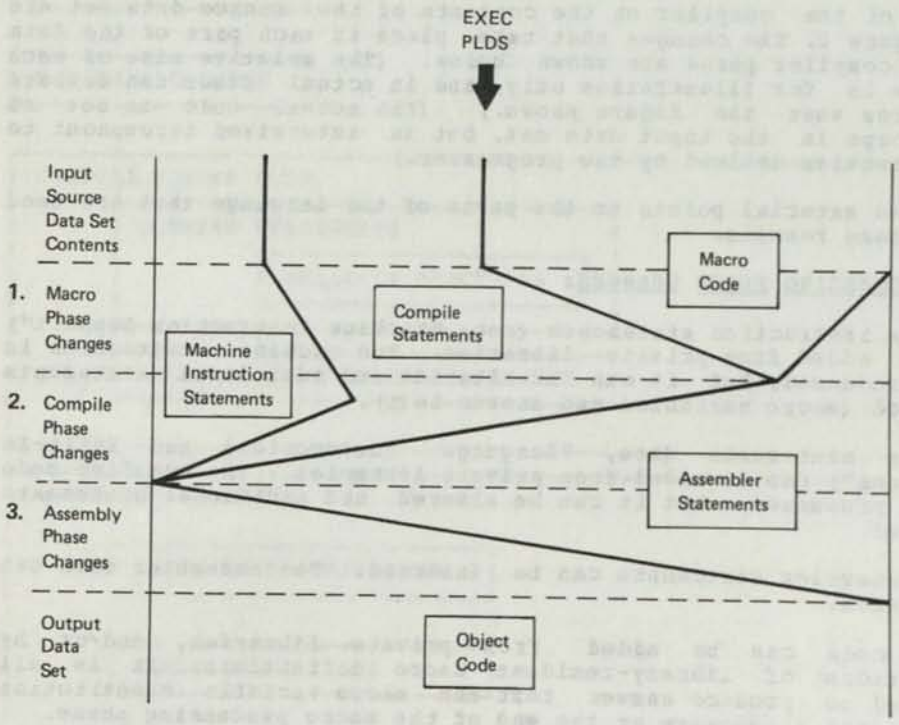


Figure 2. Phase-by-Phase PL/DS Processing of Source Data

CODING RULES

RECORD FORMAT

The logical records in the source code data set are 80 characters long. Columns 2 through 72 are the source text area of the input records, unless you submit the MARGINS compiler option to change the margins. Within the source text area there are no columns or fields with special uses. The compiler does not look at any source record columns outside of the source text area.

CHARACTER SET

The character set used for PL/DS is EBCDIC. Character data appearing in the PL/DS program is interpreted using EBCDIC character codes.

You can use all 256 EBCDIC characters within comments and character data constants. However, only a subset of the EBCDIC character set is used for any other part of a PL/DS statement. This subset consists of:

- The letters A through Z, and the special characters #, $, and @.
- The digits 0 through 9.
- The 19 special characters in Figure 3.

Letters, digits, and the characters #, $, and @, are used in forming language keywords, data constants, and variable names for data. The special characters are used, singly or in combination, either as delimiters or to form operators. Delimiters and operators formed of special characters in combination are called composite delimiters and composite operators, respectively.

Character	Name	Character	Name
	Blank	;	Semicolon
=	Equal or assignment sign	:	Colon
+	Plus or addition sign	'	Apostrophe
-	Minus or subtraction sign	¬	"Not" sign
*	Asterisk or multiplication sign	&	"And" sign
		|	"Or" sign
/	Slash or division sign	>	"Greater than" sign
(	Left parenthesis	<	"Less than" sign
)	Right parenthesis	%	Percent sign
,	Comma	?	Question mark

Figure 3. Special Characters in the PL/DS Character Set

RULES FOR CODING IDENTIFIERS

Identifiers are names that you code, such as labels of statements and names of variables. Your identifiers must conform to the following:

- One to eight characters long.
- First character can be one of the alphabetic characters A through Z, or # or $.
- Characters 2-8 can be any of the alphabetic characters, the numeric characters 0 through 9, and the special characters #, $, or @.

SOURCE STATEMENT FORMAT

The statements are free-form. They can begin anywhere in the source text area of a record, and continue on as many consecutive records as necessary without any continuation characters. (The FORMAT compiler option produces listings that have the statements neatly and logically aligned, regardless of the input record format.)

The only exception to this is the compiler control statements. The "at sign" (@) prefix must be the first non-blank character, within margins, in the record, and the @ xxxx control statement must be the only statement, other than a comment, in the record.

STATEMENT DELIMITERS

If you keep in mind how the compiler recognizes the beginnings and endings of statements, you can "think like the compiler" as you code, and avoid unintended results. Remember, a new line isn't automatically a new statement.

The usual statement-ending delimiter is the semicolon.

There is no special "begin" character for high level compile source statements. The keywords these statements contain are their identifiers. The begin characters for other PL/DS language components are shown on the syntax diagrams in their descriptions in this manual. The unusual cases are comments (described below), macro definition statements, and assembler statements. Assembler statements are set off by the special statements MACGEN and @ENDGEN before and after them to say "everything in between is assembly source."

RULES FOR USING BLANKS

Blanks can be used freely on a PL/DS statement, except for the following restrictions:

- Blanks cannot be embedded in variable names, labels, entry names, and keywords.
- Blanks cannot be embedded in data constants, other than in character constants as meaningful characters.
- Blanks cannot be embedded in composite delimiters and composite operators.

Note: Variable names, keywords, and data constants are "tokens" (smallest meaningful character groupings), and cannot be immediately adjacent to one another. Some appropriate delimiter, such as one or more blanks, must separate them.

SUBSCRIPT AND SUBSTRING EXPRESSIONS

An operand can contain a subscript or substring expression. These expressions may be any valid expression. Their evaluated value is used to arithmetically calculate the subscript or substring. The value of the subscript or substring expression must not be greater than the associated dimension (subscript) or length (substring). It must be greater than zero.

Subscript Expressions

A subscript expression is used to reference an element of an array. It gives the position of the element within the array. The expression, enclosed in parentheses, follows the variable name assigned to the array; that is, arrayname(subscript expression). For multiple dimensions there must be as many subscript expressions within the parentheses as there are dimensions. Multiple subscript expressions are separated by commas.

A subscript expression may be any expression valid arithmetically.

Examples of subscript expressions are:

```
(3)
(1)
(3 + J)
(B->C)
(I+J , K*10)
```

Substring Expressions

A substring expression is used to reference one or more bits or characters of string data. See, "String Data DECLARE Statement Attributes)," under the DECLARE statement, for more information.

To reference one bit or character, the expression, enclosed in parentheses, follows the variable name assigned to the string data, that is: variablename(substring expression).

To reference more than one bit or character, you must indicate the position in the string data of the first bit or character and the position of the last bit or character. The first position and the last position are separated by a colon.

To reference a portion of string data that is an element of an array, the substring expression is preceded by the subscript expression (or expressions separated by commas, for multiple dimensions). The substring and subscript expressions are separated by a comma. Examples of substring expressions are:

```
DECLARE STRING CHAR(100);        /* SCALAR 100 BYTES LONG                 */
DECLARE SDIM1(10) CHAR(10);      /* ARRAY OF 10 ITEMS - EACH 10 BYTES     */
DECLARE SDIM2(5,2) CHAR(10);     /* 2-DIMENSIONAL ARRAY - EACH 10 BYTES   */
   STRING(3:3)                   /* THE THIRD BYTE OF STRING              */
   STRING(3)                     /* THE THIRD BYTE OF STRING              */
   STRING(3:4)                   /* 2 BYTES STARTING WITH THE THIRD       */
   STRING(J:J+3)                 /* 4 BYTES STARTING AT JTH               */
   SDIM1(7,2:2)                  /* SINGLE BYTE FROM THE 7TH ELEMENT      */
   SDIM1(7,2)                    /* SINGLE BYTE FROM THE 7TH ELEMENT      */
   SDIM2(4,2,3:9)                /* 7 BYTES FROM ELEMENT "SDIM2(4,2)"     */
   STRING(J)                     /* A SINGLE BYTE                         */
   STRING(J:J+K)                 /* BOTH BOUNDS DYNAMIC                   */
   SDIM2(I,J,4:A+B)              /* UPPER BOUND DYNAMIC                   */
```

Variable length substrings are possible. See, "Variable Length Substrings," under the Assignment statement, for more information.

PL/DS REGISTERS

Before coding any PL/DS statements in which you must specify registers, it is important to be familiar with the register nomenclature used by the compiler, described in Appendix I of this publication. This nomenclature is consistent with that used by the DPPX Assembler.

The LINKAGE option you specify in your PROCEDURE statement also impacts your register usage.

POINTER NOTATION FOR INDIRECT ADDRESSING

Pointer notation uses an operator, ->, made up of two special characters: - and >. The format of pointer notation is:

```
pointer variable -> indirect data
```

where:

pointer variable is the name of a variable whose value is the address of the indirect data.

indirect data is the name of the data to be found at the address contained in the pointer variable.

In the following example:

```
DECLARE P POINTER;
DECLARE A CHARACTER(8);
P->A='ABCDEFGH';
```

...the variable P declared as POINTER holds the address of variable A.

RULES FOR CODING COMMENTS

You can include comments wherever a blank is permitted. Comments are enclosed by the characters /* to the left of the comment and by */ to the right of the comment. The characters /* must not appear in statement columns 1 and 2.

Comments can include any valid EBCDIC characters. However, the combination */ should not appear within a comment, because these characters terminate the comment.

SYNTAX DIAGRAMS

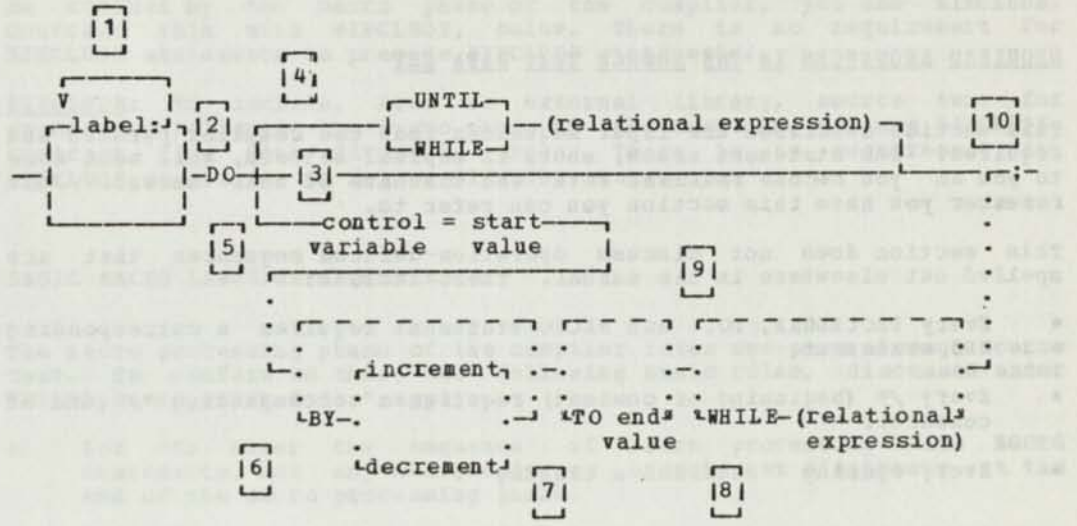
This manual uses a different language syntax notation than you have encountered in other IBM products. The syntax diagrams are words and characters connected with lines to give the required sequences for coding special characters, keywords, and various operands.

Basic Syntax Rules

- All special characters and capital letters must be coded as shown.
- Words in small letters are statement elements that you must code the appropriate values for.
- When the horizontal path (line) branches out into two or more lines, the elements presented on each of the branchout lines are optional elements to choose from at that point.
- Elements or sets of elements that you can repeat as often as appropriate have a line branching up and back to the first of them.
- Elements that you do not need to code in the sequence of their appearance in the syntax diagram are identified by broken lines leading to and from them.

Example of Syntax Rules

Figure 4 is an example of a syntax diagram, demonstrating the syntax diagram rules just described. The numbers are for this example only, and do not appear on standard syntax diagrams.



1. The label is an optional element in the statement. There can be more than one label. All labels must precede the DO keyword.

2. The DO keyword is the only element of this statement, other than the final semicolon, that must appear (see option 3).

There are three optional and mutually exclusive forms of this statement (3, 4, 5), all shown after (to the right of) the DO keyword.

3. There may be no other elements after the DO keyword.

4. There may only be a WHILE or UNTIL keyword with its associated expression after the DO keyword.

5. A control variable may be assigned a start value. This element can be optionally followed by the BY, TO, or WHILE keywords with their associated arguments. These keywords may be written in any order; it is not necessary that they be written as in the diagram.

6. The optional BY keyword may be used to specify an incrementing or decrementing value to be added to/subtracted from the control variable.

7. The optional TO keyword indicates a limit for the control variable. When exceeded, the DO processing is terminated.

8. The optional WHILE keyword must have an expression with it. The expression defines the conditions under which the DO processing will continue.

9. The broken and dotted lines show that BY..., TO..., and WHILE... may be coded in any sequence.

10. The semicolon is a special character that must be written as indicated to signify the logical end of the DO statement.

Figure 4. Explanatory Syntax Diagram

INTRA-MODULAR LINKAGE

"Simplified intra-modular linkage for programs that will be used under DPPX Base" is a PL/DS feature listed in the introduction of this publication. To use this feature, see the LINKAGE(3) option in the description of the PROCEDURE statement.

REQUIRED SEQUENCES IN THE SOURCE TEXT DATA SET

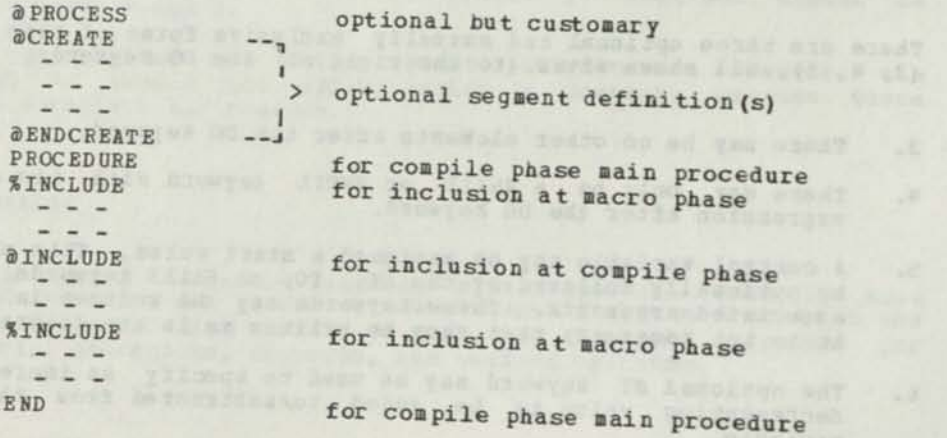
This section describes the input sequences that the compiler permits and requires. The statement names, shown in capital letters, will mean more to you as you become familiar with the contents of this manual. Just remember you have this section you can refer to.

This section does not discuss operation-defined sequences that are spelled out elsewhere in the manual. These include:

- Every PROCEDURE, DO, and MACRO statement requires a corresponding END statement.
- Every /* (beginning of comment) requires a corresponding */ (end of comment).
- Every opening ' requires a closing '.

REQUIRED SOURCE TEXT SEQUENCE

The basic sequence of statements that your source text data set must have is:



@PROCESS: This is an optional statement, described under "Compiler Options", that you can use to specify compiler options for this data set's compile.

If you submit an @PROCESS statement it must be the first record in your source data set.

@CREATE...@ENDCREATE: This is a set of statements, described under compiler control statements, that you can use to submit sets of statements (segments) that you will later include (with @INCLUDE). If you submit @CREATE...@ENDCREATE statements, you must place them at the beginning of your data set, immediately after any @PROCESS statement.

PROCEDURE...END: The main procedure of your program must start with a PROCEDURE statement and end with an END statement. This is required for the compile phase of the compiler. (You can either place these actual statements in your source data set, or make sure that they will be inserted by either @INCLUDE, %INCLUDE, or by answer text from macros you invoke.)

%INCLUDE: To include, from an external library, source text that is to be scanned by the macro phase of the compiler, you use %INCLUDE. Contrast this with @INCLUDE, below. There is no requirement for %INCLUDE statements to precede @INCLUDE statements.

@INCLUDE: To include, from an external library, source text for processing following the macro phase of the compiler, you use @INCLUDE. Contrast this with %INCLUDE, above. There is no requirement for @INCLUDE statements to follow %INCLUDE statements.

BASIC MACRO LANGUAGE SEQUENCES

The macro processing phase of the compiler takes one pass through source text. To conform to this, the following basic rules, discussed under "PL/DS Macro Outer Code," apply:

- You can alter the sequence of macro processing with %GOTO statements, but any macro code you branch past disappears at the end of the macro processing phase.
- Macro code that alters source code strings must be processed before the strings it alters.
- Macro code that issues answer code must be processed at the position that the answer text is to appear.
- Macro definitions that are included in this data set must appear before their invocations are processed.
- Source text that is on separate data sets and that is to be processed in the macro phase of the compiler must be brought into the source stream with a %INCLUDE statement at the place where it is to appear.

COMPILER OPTIONS AND CONTROLS

When you submit a job for compilation, there is a set of compiler options and compiler controls to choose from.

The compiler options that you choose are in effect for the whole compile job. You intersperse the compiler controls with your source statements wherever you need them.

In the category of compiler options and controls, there is a test option for use by coders of macro definitions. See the description of VBTBC, under "Macro Definition Debugging Aids."

COMPILER OPTIONS

The compiler options are explained in additional detail in Appendix H of this manual, and in full detail in the publication, Programming Language for Distributed System (PL/DS), User's Guide. You can either omit compiler options and accept the default values for them, or specify the options with the values that you want. You can specify options either in the PARM field of the EXEC statement or in an @PROCESS compiler control statement at the first of your source statements. When an option is in both the PARM field and the @PROCESS statement, the PARM field specification is used by the compiler.

ADECK: Write object deck resulting from assembly phase to SYSPUNCH.

ADEFS: Define all variables in assembler code.

ALIST: Print assembly phase listing.

ANNOTATE: Show compiler source statements in assembly phase listing.

ASSEMBLE: Perform assembly phase, conditional on level of error encountered in compile phase.

ATITLE: Specify identifier for assembly listing pages and for SYSLIN (see OBJECT option) object code records.

AXREF: Produce assembly cross reference listing.

BUFSIZE: Specify amount of INCLUDE buffer storage.

COMPILE: Perform compile phase.

ESD: Produce listing of external symbol dictionary.

EXTEND: Increase static table storage for OPTIMIZE.

FDECK: Write formatted source to SYSPUNCH.

FLAG: Specify severity level of messages to print on listing.

FORMAT: Indent and align structured code on listing for readability.

FSOURCE: Print unformatted source on listing.

IDR: Add CSECT information data record to each compiler produced external procedure END statement.

LINECOUNT: Number of lines per listing page.

MACPARM: Pass data to the macro phase.

MACRO: Perform macro processing phase.

MARGINS: Specify margins of input records that contain compiler source.

MDECK: Punch macro processing phase output to SYSPUNCH.

MPERCENT: Amount of SIZE option for global dictionary and string area.

MSOURCE: Print original source, before any macro processing, on listing.

MXREF: Print macro variable attribute and cross reference.

OBJECT: Format object code for simulator (SYSLIN), or for transfer to DPPX (SYSITEXT), or both.

OPTIMIZE: Optimize the code produced by the compile phase.

OPTIONS: Print compiler options at beginning of listing.

RLD: Print listing of the relocation dictionary.

SIZE: Specify dictionary and text buffer size.

SOURCE: Specify printing of source.

STATISTICS: Print compile phase statistics, such as number of statements.

TERMINAL: Specify message severity for SYSTERM.

TEST: Produce special source symbol table (SYM) object module records.

TITLE: Specify heading for each page of compile phase listing.

XREF: Print attribute and cross reference table on listing.

COMPILER CONTROL STATEMENTS

Besides the special PROCESS control statement described above for specifying compiler options, there are six other control statements.

As described above, under "Source Statement Format," there can be no other statement except a comment on a compiler control statement line. The @ sign prefix must be the first non-blank character after the left margin of the first line of a compiler control statement. Each statement can take as many additional lines as required to complete.

@CREATE: Begins and names a segment for later inclusion with an @INCLUDE or %INCLUDE statement.

@EJECT: Specifies the conditions for an eject to a new printout page at this place in the compiler-produced listing.

@ENDCREATE: Ends the segment started with the last @CREATE statement.

@INCLUDE: Incorporates compiler source statements from external libraries into this compilation.

@LIST: Allows changing, overriding, and restoring the control for printing compiler-generated lines after this place in the program. This applies to the listing after the compile phase.

@PROCESS: Specify compiler options. (Note: the PROCESS statement usage is described in the section, "Compiler Options.")

@SPACE: Leave blank line(s) on listing.

CHAPTER 2. COMPILE STATEMENTS AND BUILT-IN FUNCTIONS

This section describes the rules for coding compile statements and the built-in functions that the compiler supports for these statements.

In addition to these statements, you can code machine instructions, as described in the section, "Built-In Machine Instructions."

Besides in-line source code, you can bring the contents of source code data sets into your input.

Your set of source statements can be dynamically altered or added to during a compile run using the macro language. A macro phase of the compiler acts on your macro instructions and performs the specified alterations before the compile phase processes the final source statements. See the section, "Compiler Macro Language."

COMPILE STATEMENTS

SUMMARY OF COMPILE STATEMENTS

STATEMENT	USE
assignment	Dynamically assigns source expression value to variable.
CALL	Invokes an external or internal procedure.
DECLARE	Describes attributes of a variable.
DO	Starts a do-group.
ELSE	States action when IF condition is false.
END	Ends do-group or end of procedure.
ENTRY	Marks secondary entry points in a procedure.
MACGEN, ENDGEN	Identifies assembler text to be passed unchanged through compile phase.
GOTO	Transfers control (normally within procedure).
IF	States condition for IF, THEN, and (optionally) ELSE sequence.
null statement	Indicates no compiler action (fall through), usually in a THEN or ELSE.
PROCEDURE	Marks primary entry point in procedure.
RESPECIFY	Overrides attributes of variable.
RETURN	Returns control from a called procedure.
THEN	States action when IF condition is true.

Assignment -- Assign a Value To a Variable

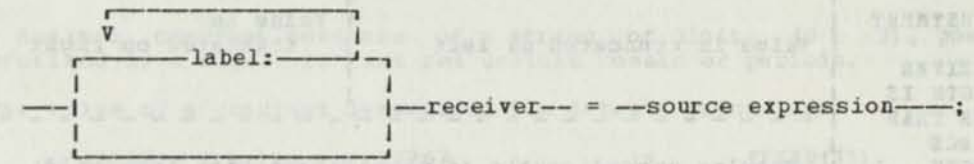
Purpose

Assign a value to all or part of a variable, changing the value specified in the DECLARE statement INITIAL attribute.

Rules

Only the receiver variable is changed.

Syntax



Operands

label	Optional. Code one or more labels, each followed by a colon.
receiver	A variable that is to be assigned a value. It must be a variable declared FIXED BINARY, CHARACTER, BIT, or POINTER. If the receiving variable describes an element of an array, you will also code a subscript expression. If the receiving variable describes part of a character or bit string, you will also code a substring expression. If the receiving variable is BASED, you can explicitly pointer-qualify it. For coding these special cases, see "Coding Rules," in Chapter 1.
source expression	Simple item (a single operand) or a complex expression (one or more operands and one or more operators) whose result the compiler assigns to the receiver.

Operations Peformed in the Assignment Statement

The compiler performs the assignment with a pointer, arithmetic, or string operation based on the following algorithm:

```
IF receiver is POINTER or FIXED(32)
THEN
   pointer assignment
ELSE
   IF receiver is FIXED or source is FIXED
   THEN
      arithmetic assignment
   ELSE
      string operation
```

When the receiver and source have different lengths, the compiler adjusts the assigned value in the receiver as in Figure 5.

OPERATION TYPE —> (see algorithm)	POINTER	ARITHMETIC		STRING	
OPERAND CHARACTER -ISTICS —>	Source must be unsigned*	Source is signed	Source is unsigned	Source and/or receiver are BIT	Source and receiver are CHARACTER
ADJUSTMENT IF RECEIVER LENGTH IS GREATER THAN SOURCE LENGTH —>	Value is extended on left with zeros	Value is extended on left with sign	Value is extended on left with zeros	Value is extended on right with zeros	Value is extended on right with blanks
ADJUSTMENT IF RECEIVER LENGTH IS LESS THAN SOURCE LENGTH —>	Value is truncated on left			Value is truncated on right	
	(Results unpredictable if significant bits truncated)				

* If source is signed, it is treated as unsigned.

Figure 5. Adjustments to Length of Assigned Value

Notes on BIT Receivers:

The following requirements exist when the receiver is BIT:

1. The length of the receiver and its starting bit position must be known at compile time.

2. The length of the source and its starting bit position must be known at compile time.

3. The length of the source must equal the length of the receiver if the source is a variable.

4. If either of the source or receiver bit strings crosses two or more byte boundaries, then either the source must be a bit constant, or both source and receiver must be multiples of eight bits aligned on a byte boundary.

5. If the source is not a bit variable, or a bit string constant, or any expression other than "¬(bit-variable)", then both source and receiver must be multiples of eight bits aligned on a byte boundary.

Notes on String Receivers:

The following length requirements exist for string receivers:

1. A string receiver that is an eight bit multiple aligned on a byte boundary cannot be longer than 256 bytes.

2. A bit string receiver that is not an eight bit multiple aligned on a byte boundary cannot be longer than 32 bytes.

COMPLEX SOURCE EXPRESSIONS IN ASSIGNMENT STATEMENTS

Operands in the Source Expression

Source expression operands can be either variable names or constants. Variables may be FIXED, POINTER, CHARACTER, or BIT. If a variable describes an element of an array, the variable will include a subscript expression. If a variable describes part of a character or bit string, the variable will include a substring expression. If a variable is BASED, it can be explicitly pointer qualified.

Constants may be arithmetic or string; how to code each type of constant is described below.

Arithmetic Constants

There are two types of arithmetic constants: decimal and binary.

A decimal constant consists of a string of digits (0 - 9), possibly prefixed by a sign. It must not contain commas or periods.

The value of a constant determines its default precision:

```
-32768 = < value = < 32767              is      FIXED(15)
 32768 = < value = < 65535              is      FIXED(16)
 65536 = < value = < 2147483647         is      FIXED(32)
```

A binary constant consists of a string of digits 0 and 1, followed by the letter B (for example, 100B is equal to 4); the constant may be signed or unsigned. When the constant is converted into decimal form, its value must conform to the rules for decimal constants.

String Constants

There are three types of string constants: character, bit, and hexadecimal.

A character constant consists of any EBCDIC characters enclosed in apostrophes; if any of the characters is an apostrophe, code two consecutive apostrophes.

A bit constant consists only of the digits 0 and 1 enclosed in apostrophes and followed by the letter B.

A hexadecimal constant consists of any hexadecimal digits (0-9, A-F) enclosed in apostrophes and followed by the letter X. It is a shorthand for the equivalent bit constant. For example, the hexadecimal constant 'B'X is the same as the bit constant '1011'B.

The null character string is coded as ''; the null bit string is coded as ''B. For examples of null strings that set areas to zeros or blanks, see the section, "Coding Common Functions," in Appendix G.

The maximum length of a character, hexadecimal, or bit constant is 256 input characters. To calculate the length of any constant you count the number of input characters, excluding the enclosing or doubling apostrophes. For example:

- The character constant 'O''NEIL' counts as six characters.
- The hexadecimal constant 'B'X counts as one character. Note that its bit configuration is the same as that of the bit constant '1011'B, which counts as four characters.
- The bit constant '010'B counts as three characters.

Operators in the Source Expression

Figure 6 shows the operators that can be used in a source expression.

OPERATOR	OPERATION	PRIORITY	ASSOCIATED DATA TYPE
+	Prefix plus	1	Arithmetic or pointer
¬	Negation	1	Bit
-	Prefix minus	1	Arithmetic or pointer
*	Multiplication	2	Arithmetic
/	Division	2	Arithmetic
//	Remainder	2	Arithmetic
+	Addition	3	Arithmetic or pointer
-	Subtraction	3	Arithmetic or pointer
&	AND	4	Bit or arithmetic
|	Inclusive OR	5	Bit or arithmetic
&&	Exclusive OR	6	Bit or arithmetic

Note: When -> (pointer variable notation), or subscripting, or substringing, or any of the built-in functions appears in the source expression, it is treated first.

Figure 6. Table of Expression Operators

In complex expressions, the order of associating operands with operators in the expression is determined by the priority of each operator. The operands connected by the highest priority operator (lowest number) are associated first, the operands connected by the second highest priority operator are associated next, and so on.

If more than one operator of priority 1 appears in the expression, association of their operands is done from right to left. If more than one operator of the same priority -- other than priority 1 -- appears in the expression, association of their operands is done from left to right.

If you want to change the order of association, you may use parentheses. Operators and operands enclosed in parentheses are associated first and are associated by priority. For example, in this statement:

```
A = 2*(3+4);
```

...(3+4) is associated first; the result is multiplied by 2. In the statement

```
B = M*(N*(Q/P-Q));
```

...(Q/P-Q) is associated first; the result is multiplied by N; the result of multiplication is multiplied by M.

Operation Sequence in a Complex Source Expression

If the source expression is complex, a pointer, arithmetic, or string operation is performed for each operator in the expression. Which type of operation is performed at a given stage of the evaluation depends entirely on the operator and the attributes of its operands. The attributes of intermediate results are described below.

Pointer Operations

A pointer operation is performed when the operation is addition or subtraction and one or both operands is a pointer. The result is an unsigned value with precision 32 of type POINTER.

If the operands have different precisions, the shorter operand is conceptually extended on the left: with its sign if the item is signed, or with zeros if it is unsigned. String operands are allowed if they are byte-aligned and one or two or four bytes in length; they are treated as FIXED(8) or FIXED(16) respectively.

Pointers may only be used as operands of addition, subtraction, prefix addition, or prefix subtraction.

Arithmetic Operations

An arithmetic operation is performed if one or both operands are arithmetic and neither is pointer. The result after applying any arithmetic operators is usually a halfword arithmetic value. There is one exception: if either operand is FIXED(32), the result is FIXED(32).

If the operands have different precisions, the shorter operand is conceptually extended on the left: with its sign if the item is signed, or with zeros if the item is unsigned. String operands are allowed if they are byte-aligned and either one or two bytes long; they are treated as FIXED(8) or FIXED(16), respectively. If either operand is FIXED(16), the result is UNSIGNED; if neither operand is FIXED(16), the result is SIGNED.

See Appendix D for tables showing precision rules.

Restrictions on Division and Remainder Operations

There are two restrictions on division (/) and remainder (//).

1. Both dividend and divisor must be unsigned.
2. The divisor cannot have precision greater than 16.
3. The divisor cannot be 0.

The result is unsigned, of precision 16.

If both operands are constants, signed division with precision of 31 is performed.

String Operations

A string operation is performed if both operands are strings and the operation is "Or," "Exclusive Or," or "And." Both operands must have the same length, and must be eight-bit multiples aligned on byte boundaries. A character is considered as eight bits.

The result is a string that has the same length as the operands.

Variable-Length Substrings

The following rules apply for variable-length substrings:

1. A variable-length substring in a source expression must not be longer than the receiver.
2. A variable-length substring that is the receiver must not be longer than the source.
3. A variable-length substring must not exceed the range of the string data of which it is a portion.

4. A variable-length substring cannot be used as an arithmetic operand.

5. If the receiver of an assignment statement is a variable length substring, no padding of blanks or binary zeros is done.

Caution: The compiler assumes that the above rules have been followed.

A bit substring expression that references one bit can only be a constant value.

If a bit string has a variable substring range, the compiler assumes -- but does not check to ensure -- that: (a) the lower bound specifies the first bit of a byte and that (b) the upper bound specifies the last bit of a byte.

CALL -- Invoke an External or Internal Procedure

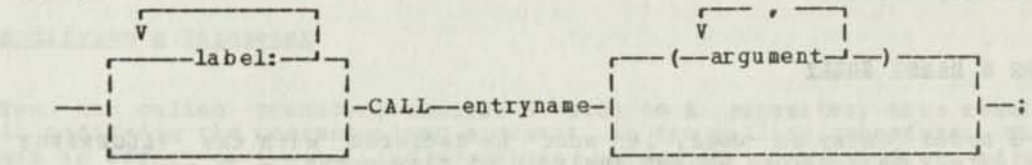
Purpose

You use CALL to cause the compiler-generated code to invoke another procedure during execution.

Rules

CALL is a reserved keyword. Do not use it as the name of a variable.

Syntax



Operands

label — Optional. Code one or more labels, each followed by a colon.

entryname — Name of a PROCEDURE statement or of an ENTRY statement, or of a name declared with the attributes BASED and ENTRY. See the BASED attribute of the DECLARE statement. The entryname must appear in an internal procedure of the calling procedure, or in an external procedure. The entryname cannot be in a procedure that is internal to either an internal procedure in the calling procedure or a separately compiled procedure.

argument — Code none, one, or more parameters, separated by commas, to pass to the called procedure. Each one must be a constant, a variable, or an expression.

CALL Arguments

You can pass up to 16 arguments, each separated by a comma, to the procedure you are calling. An argument can be a constant, a variable, or a complex expression. The two restrictions for arguments are:

1. When your argument is a REGISTER variable, the compiler issues a warning message, and assigns the value to a compiler-created temporary variable in storage.
2. When your argument is a bit string that is not on a byte boundary, the compiler issues a warning message.

Some examples of valid arguments are:

```
A           ARRAY(3)
32          TAB(M)
C*D         (LIST(4))
(J)         ITEM(7,2:5)
```

There is an example of a CALL statement in the description of the DECLARE statement SETS/NOSETS entry option.

Declaring the Entry Name of a Called Procedure

In the calling and called procedure, you need not declare the entry name as ENTRY. If you do not declare the entry name, appropriate defaults are assigned when the entry name is encountered. For a detailed description of how to declare an entry name refer to the discussion of the DECLARE statement.

The declare statement for the entry name should appear in the calling procedure. It need only include the entry name and any selected options. There are ten options that may be used:

FLOWS/NOFLOWS	EXIT/NOEXIT	REFS/NOREFS
SEQFLOW/NOSEQFLOW	SETS/NOSETS	

They are explained in the description of the DECLARE statement ENTRY attribute, below.

Using a Based Entry

If a based entry is used, it must be declared with the VALUERANGE attribute. This allows proper analysis of flow paths.

Example of CALL Using a Based Entry

```
DCL RTN BASED VALUERANGE(RTN1,RTN2,RTN3) ENTRY,
    RPTR(3) PTR INIT(ADDR(RTN1),ADDR(RTN2),ADDR(RTN3));

CALL RPTR(I)->RTN;
```

Alternatively, if RTN is declared BASED(RPTR(I)), the CALL statement can be simplified to:

```
CALL RTN;
```

How the CALL Arguments are Passed

When you include arguments on a CALL statement, a parameter list is produced. The parameter list contains one word for every argument. The address corresponding to an argument is inserted in this word. Three possible kinds of addresses are inserted, depending on the type of argument.

1. Variables: The actual address of a single variable is inserted in the parameter list. The variable can be subscripted or substringed, or explicitly or implicitly pointer qualified. A bit variable not on a byte boundary will use the byte address of the first bit.

When a REGISTER variable is specified, the value of the register will be stored in a compiler-generated temporary, and the address of the temporary will be used.

2. Constants: For arguments that are constants, the value of the constant is assigned to storage and the address of the assigned storage area is inserted in the parameter list.

3. Complex Expressions: The expression is evaluated and the result is assigned to a compiler-generated temporary variable. The address of the temporary variable is inserted in the parameter list.

Argument and Parameter Correspondence

The arguments on a CALL statement correspond, by their positions in the list, to the parameters on the PROCEDURE or ENTRY statement you are calling. The compiler associates arguments with parameters in the order

in which they appear: the first argument is associated with the first parameter, the second argument with the second parameter, and so on.

Since the association of arguments and parameters is by order, it may not matter whether the variable names used are the same or different. However, when you are passing arguments to an internal procedure, the variable names used for the arguments must be different from the names used for the parameters.

Declaring Arguments and Parameters

Arguments are declared in the calling procedure; parameters are declared in the called procedure. The data type attributes you declare for a parameter should be the same attributes you declare for the corresponding argument in the calling procedure.

Modifying a Parameter

When the called procedure assigns a value to a parameter, this results in modifying the corresponding argument in the calling procedure. This has no effect on the calling procedure unless the actual address of the argument was inserted in the parameter list or the argument was a constant. (Which arguments have their actual address inserted in the parameter list is described above in "How the CALL Arguments are Passed.") You should not modify a parameter whose corresponding argument is a constant, since this could cause unpredictable results in the calling procedure.

> Note: To guarantee evaluation and assignment of an argument to a temporary variable, you must code the EVAL built-in function. For example, EVAL can be used to protect the CALL argument from changes by its corresponding parameter.

Where Control is Returned

Control is normally returned to the statement immediately following the CALL statement. Control is returned when either a RETURN statement or a procedure END statement is encountered. If you want control returned to some other point within the calling procedure, use the RETURN TO option of the RETURN statement. You should identify other return point names by using the FLOWS option when declaring the entry name.

> Note: If you want a return code value returned in register 2, use the CODE option of the RETURN statement. (If LINKAGE(3) is specified, the value is in register 30. This is the convention to use in a program that is to run on DPPX.)

DECLARE -- Name a Variable and Assign its Attributes

> Note: There are two other forms of the DECLARE statement. See, "DECLARE -- Specifying a Structure," and "DECLARE -- Specifying an Array."

Purpose

States, to the compiler, the names and attributes of the variables and named constants in your program.

Rules

When using individual DECLARE statements to build a structure, the statements must be grouped together.

When you omit specifying one or more of the attribute types, the compiler assigns a default value for those attribute types. (See Appendix A.)

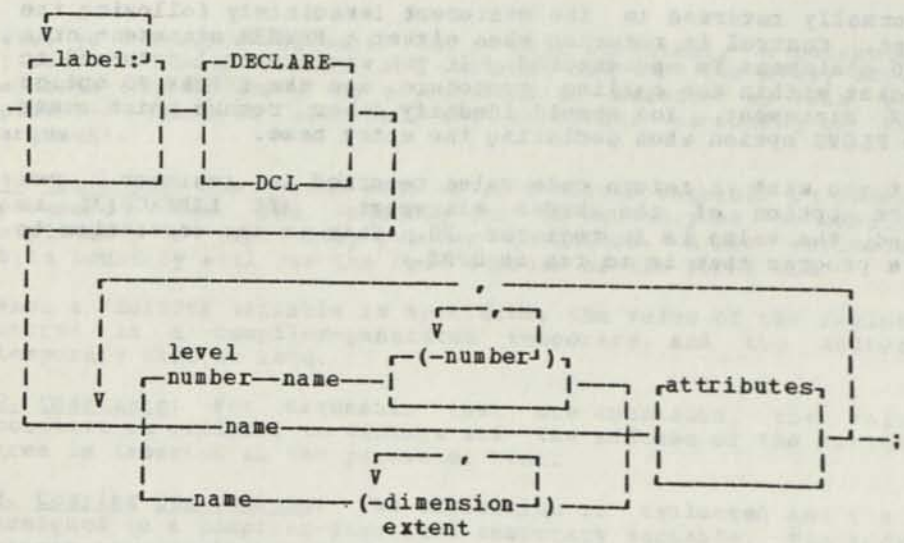
When you use a variable that you have not declared, the compiler assigns a set of attributes.

See the assignment statement for specifying the value of a variable.

Each variable name declared in the program must be different, to represent only one variable, even in disjoint procedures (procedures that are not internal or external one to the other).

DECLARE and DCL are reserved keywords; do not use them as variable names.

Syntax



Operands

label Optional. Specify one or more labels, each followed by a colon.

level number For a data item within a structure, give the item's level. See also, "DECLARE -- Specifying a Structure."

name — Specify the name of the variable you are declaring.

number — For an array, give the number of identical items in the array. For a multi-dimensioned array, separate the dimensions with commas. See also, "DECLARE -- Specifying an Array."

attributes — Specify the seven attribute types of the variable. The attribute types and their corresponding values are described under "Attribute Types and Their Options for the DECLARE Statement," below. Code them separated with blanks.

Declaring More Than One Variable on a Statement: To declare more than one variable on the same DECLARE statement, separate each declaration with a comma. For example:

```
DCL BASE POINTER,
AREA CHAR(12),
COUNT FIXED(15);
```

Simplified Coding of Variables With Common Attributes: Attributes common to several variables can be factored to eliminate repeated specification. To use factoring, place in parentheses the variable names separated by commas; follow each name with any unique attributes. List, following the right parenthesis, those attributes that the variables have in common. For example:

```
DCL (FLAGA,FLAGB) BIT(1);
```

...FLAGA is declared as BIT(1) and FLAGB is declared as BIT(1).

In the following:

```
DCL (B BIT(8), C CHAR(20)) EXTERNAL;
```

...B is declared as BIT(8) and EXTERNAL. C is declared as CHAR(20) and EXTERNAL.

Partial declarations may themselves involve factoring. For example:

```
DCL (LV FIXED(15), (SB BIT(1), SC CHAR(16)) EXTERNAL) LOCAL;
```

...LV is declared as FIXED(15) and LOCAL. SB is declared as BIT(1), EXTERNAL, and LOCAL. SC is declared as CHAR(16), EXTERNAL, and LOCAL.

When declaring a structure, the level number can be factored to the left of the factoring parenthesis. For example:

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

...This declares B, C and D as level twos of A.

ATTRIBUTE TYPES AND THEIR OPTIONS FOR THE DECLARE STATEMENT

The table in Figure 7 summarizes the seven types of attributes and their options for the attributes operand in the DECLARE syntax. Descriptions of each attribute follow.

DATA TYPES	STORAGE CLASS	BOUNDARY ALIGNMENT	INITIAL -IZATION	NORMAL -ITY	RESTRICTION	SCOPE
Arithmetic	AUTOMATIC	BOUNDARY	INITIAL	NORMAL	RESTRICTED	INTERNAL
FIXED		BYTE		ABNORMAL	UNRESTRICTED	EXTERNAL
SIGNED	BASED	HWORD				
UNSIGNED	POSITION	WORD				
BINARY	DEFINED					
SIGNED	POSITION					
UNSIGNED						
	REGISTER					
String	STATIC					
CHARACTER	LOCAL					
BIT	NONLOCAL					
	STRONG					
	WEAK					
POINTER						
	CONSTANT					
Program	Parameter					
LABEL*						
ENTRY*						

* LABEL or ENTRY sometimes has VALUERANGE. ENTRY sometimes has options.

Figure 7. Table of DECLARE Attribute Types and Options For Each

Data Type Attributes in the DECLARE Statement

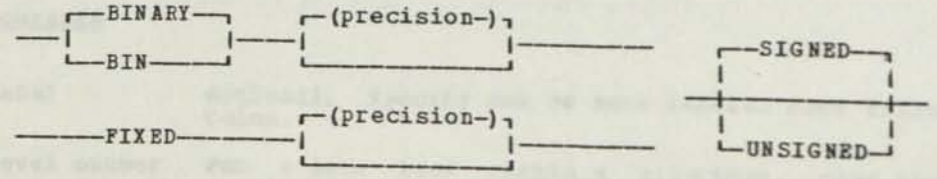
There are four different data type attributes: arithmetic, string, pointer, and program.

The data type attribute you specify determines how the bit configuration of an item is interpreted. Certain keyword attributes correspond to each type. If you do not specify a data type attribute, the data is assigned a default of FIXED(15).

See Appendix D for the precision associated with FIXED, CHAR, and BIT variables used as operands in arithmetic expressions.

Arithmetic Data Type DECLARE Statement Attributes

Arithmetic data is interpreted as a binary fixed point integer. To declare a variable as arithmetic, you code one or both of the keyword attributes BINARY and FIXED. To specify that an arithmetic variable is signed or unsigned, you can include the keywords SIGNED or UNSIGNED. The format of these attributes is:



Precision: The optional values in parentheses describe the precision of the data. The precision determines the number of bits assigned for the maximum positive value of the data. For precision 15, an additional bit is assigned as a sign bit; for a negative value, the total field represents a value in twos complement form. Precision may be specified either with BINARY or with FIXED. If you specify FIXED and BINARY, you may include a precision value following either keyword, but not both. Signed data can have a precision of 15; unsigned data can have a precision of 8, 16, or 32. The precision you specify determines how many bytes are assigned to contain the value.

Precision	Number of Bytes Assigned
8	1
15	2
16	2
32	4

Default Precision: If you do not specify a precision following the keyword BINARY or FIXED, the default precision is 15.

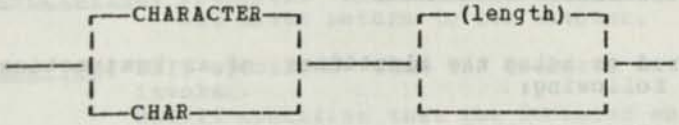
Boundary: If you do not specify boundary information (BOUNDARY attribute) when you declare the arithmetic data, boundary alignment is as follows:

Precision	Boundary
8	byte
15	halfword
16	halfword
32	word

String Data Type DECLARE Statement Attributes

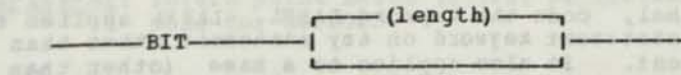
String data is divided into two categories: character and bit.

CHARACTER String Data: This is interpreted as a sequence of 8-bit EBCDIC characters. To declare a variable as a character string, you code the keyword CHARACTER. The format of this attribute is:



The length you indicate is the number of characters in the string. You may specify an asterisk (*) if the length is not known, such as for NONLOCAL data (see the STATIC storage class attribute of the DECLARE statement), or for parameters, or for based strings. When you use the asterisk, code a substring expression when referencing the string, unless the reference to the string is a CALL argument or the argument of the ADDR function. If you omit the length, a length of 1 is assumed.

BIT String Data: This is interpreted as a sequence of bits. To declare a variable as a bit string, you code the keyword BIT. The format of this attribute is:



The length you indicate is the number of bits in the string. If the length is not known, such as for NONLOCAL data or parameters or based strings, an (*) may be specified. When the asterisk is used, a substring expression is required when referencing the string, unless the reference to the string is as a CALL argument or as the argument of the ADDR function. If the length is omitted, a length of 1 is assumed.

Default Boundary: If you do not specify the boundary attribute when you declare the string data, the compiler-generated boundary alignment is as follows:

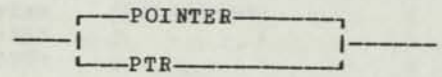
Data Type	Boundary
CHARACTER	byte
BIT (not in a structure)	byte
BIT (in a structure)	bit

Length Specification: The length field for CHARACTER or BIT items may be any expression that the compiler can evaluate at this point in the compile phase. In particular, it may use references to CONSTANT items or the LENGTH built-in function to refer to items of known length.

POINTER Data Type DECLARE Statement Attribute

Pointer data is interpreted as being a variable that contains the address of some other data. The name declared for this other data is assigned the BASED attribute. (For a discussion of the BASED attribute, see the topic "Storage Class Attributes" later in this chapter.) Pointer data is always a non-negative value.

To declare a variable as a pointer, you code the keyword POINTER or PTR. The format of this attribute is:



A pointer is an unsigned item, occupying four bytes. It is aligned on a word boundary, unless this is overridden by the BOUNDARY attribute.

An undeclared variable defaults to POINTER when it is the pointer variable operand in a pointer expression.

Program Data Type DECLARE Statement Attributes

Program data is interpreted as being the identifier of an instruction. This can be either of the following:

- An entry name on a PROCEDURE or ENTRY statement.
- A label name on any other type of statement.

You use two keyword attributes to specify program data: ENTRY and LABEL. The ENTRY attribute indicates the address of an entry name. The LABEL attribute indicates the address of a label name. A third keyword attribute, VALUERANGE, is required for a BASED ENTRY or LABEL.

Declaring a LABEL Name Program Data Type Attribute

To declare a name as a label, code the keyword LABEL. LABEL applies to a name that precedes the statement keyword on any statement other than a PROCEDURE or ENTRY statement. It also applies to a name (other than a name declared as ENTRY) that follows the keyword GOTO.

Default: A name defaults to LABEL in the following two cases:

1. When a GOTO statement's target is the name.
2. When you code the name as a label on any statement except a PROCEDURE or ENTRY statement.

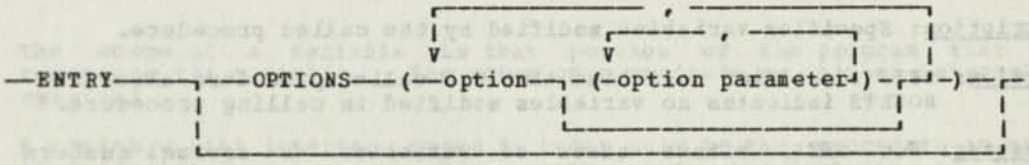
Declaring an ENTRY Name Program Data Type Attribute

To declare a variable as an entry name, code the keyword ENTRY. ENTRY applies to names that precede the keywords PROCEDURE or ENTRY. It also applies to a name that follows the keyword CALL. ENTRY is a reserved keyword; do not use it as a variable name.

Default: A name defaults to ENTRY in the following cases:

1. When a CALL statement's target is the name.
2. When you code the name immediately before either of the keywords PROCEDURE or ENTRY.

Syntax



ENTRY Options: When declaring an entry name, you may indicate one or more options. The options are specified in the OPTIONS attribute. Their function is to define the actions of the entry when it is called.

Options are available to describe an ENTRY. They are described below. Two opposite options of the same yes-no pair cannot appear within the same OPTIONS attribute.

EXIT/NOEXIT Entry Option

Description: Indicates that the declared entry may give up control and and never return to its invoker.

Function: EXIT specifies that the declared entry may not return to its invoker.
NOEXIT specifies that the declared entry always returns to its invoker.

FLOWS/NOFLOWS Entry Option

Description: Defines return point(s), other than the normal one following the call, in the calling procedure.

Function: FLOWS identifies additional return points.
NOFLOWS indicates no additional return points.

Parameter: Return point names (a list of labels) for FLOWS.
None for NOFLOWS.

REFS/NOREFS Entry Option

Description: Specifies variables referenced in the called procedure.

Function: REFS identifies items referenced in called procedure.
NOREFS indicates no references to items in called procedure.

Parameter: Either names of variables or decimal numbers representing argument positions for REFS.
No parameters for NOREFS.

SEQFLOW/NOSEQFLOW Entry Option

Description: Indicates that the declared entry may return directly to the next sequential statement following its invocation.

Function: SEQFLOW specifies that the declared entry may return to the next sequential statement.
NOSEQFLOW specifies that the declared entry never returns to the next sequential statement.

SETS/NOSETS Entry Option

Description: Specifies variables modified by the called procedure.

Function: SETS identifies variables that the called procedure alters.
NOSETS indicates no variables modified in calling procedure.

Parameter: For SETS, either names of variables or decimal numbers representing argument positions.
For NOSETS, no parameter.

Function: NOSETS indicates no variables modified in called procedure.

Example of ENTRY Attribute Option in DECLARE Statement

The following is an example of how to use the above options:

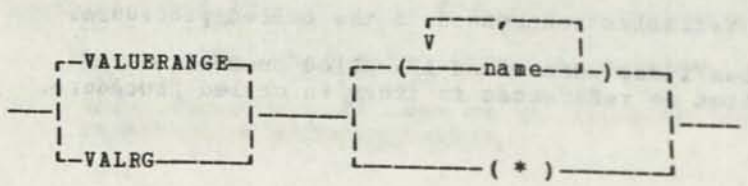
```
EXAMP: PROCEDURE;              /*CALLING PROCEDURE      */

    R
    |
DCL X LOCAL EXTERNAL;          /*REFERENCED VARIABLE    */

DCL CALLEDPR ENTRY OPTIONS(REFS(X,1,2,3),SETS(2,3),EXIT);

                               /*CALLED PROCEDURE       */

CALL CALLEDPR (A,B,C);         /*THREE VARIABLES PASSED*/
               | | |
             R R R
               S S
```

...where R is a variable referenced in CALLEDPR procedure, and S is a variable modified in CALLEDPR procedure.

Note: SEQFLOW is a default option. SEQFLOW and NOFLOWS indicate that return to EXAMP from CALLEDPR is to the statement immediately after the CALL statement; EXIT indicates that CALLEDPR might not necessarily return to EXAMP, such as in the case of an abnormal termination.

Using VALUERANGE for LABEL or ENTRY Attributes

To show possible actual values for a BASED ENTRY or LABEL, you must code the VALUERANGE attribute. The syntax of this attribute is:



name Name must be a variable declared with the LABEL or ENTRY attribute.

* When asterisk is coded, the compiler assumes that the LABEL or ENTRY variable is external to the procedure, and that you do not know the names of the actual values.

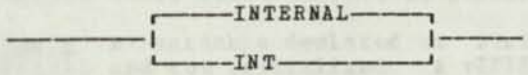
VALUERANGE has two functions:

- It supplies information required for correct code generation by the OPTIMIZE compiler option.
- It documents the possible values of the LABEL or ENTRY variable.

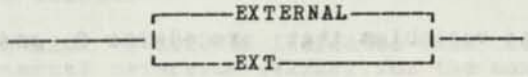
INTERNAL and EXTERNAL Attributes

The scope of a variable is that portion of the program that can reference the variable. The scope of a variable is either internal or external.

A variable with internal space is known only to the procedure in which it is declared and any procedures internal to the declaring procedure. To declare a variable as having internal scope, code the keyword:



A variable with internal scope is known to the procedure in which it is declared, and (a) to procedures internal to the declaring procedure, or (b) to separately compiled procedures (and their internal procedures) that declare the variable with the EXTERNAL attribute. To declare a variable as having external scope, code:



Default: If you do not specify a scope attribute, a variable defaults to INTERNAL, except for two cases.

1. An entry name on the outermost PROCEDURE statement or on an ENTRY statement in the outermost procedure defaults to EXTERNAL.
2. The target of a CALL statement that is either declared as BASED, or is not an entry name for an internal procedure, defaults to EXTERNAL.

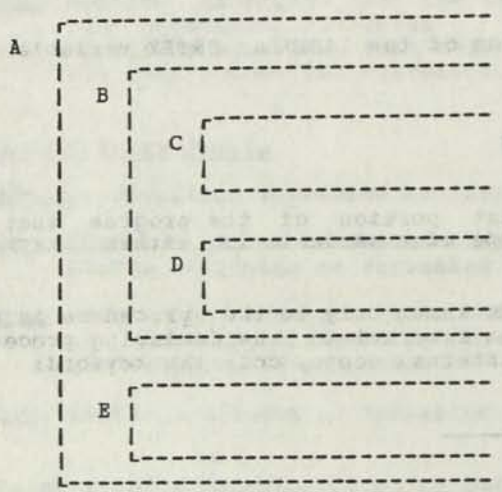
Referencing Variables Across Procedures

In a one-procedure program, all variables declared in the procedure can be referenced in the procedure. Normally, these variables cannot be referenced by any other external procedure. If you want another procedure to be able to reference one of these variables, you can do one of three things:

1. Declare the variable as EXTERNAL in both procedures. If you declare a variable in two external procedures and specify the attribute EXTERNAL, the variable may be referenced in either procedure. (If EXTERNAL is not specified, the compiler assumes that you are declaring two different variables.) The variable is assigned storage only once; you must indicate to the compiler in which procedure it should assign this space. Code the storage class attribute LOCAL when storage is to be assigned in this procedure. Code the storage class attribute NONLOCAL when storage is not to be assigned in this procedure.

2. Pass the variable as an argument on a CALL statement. Refer to "The CALL Statement" for details on how to pass arguments.

3. Use Nested Procedures. Redesign the program so that one of the external procedures is an internal procedure of the other. Any variable that is declared in a procedure can be referenced by any procedures internal to it. Once the variable is declared, it must not be declared again. Figure 8 illustrates which variables may be referenced by a procedure.



Procedure A can reference those variables that it and another external procedure declare as EXTERNAL.

Procedure B can reference any variables that procedure A declares or uses.

Procedure C can reference any variables that procedures B and A declare or use.

Procedure D can reference any variables that procedures B and A declare or use.

Procedure E can reference any variables that procedure A declares or uses.

To ensure that any internal procedure can reference a variable, declare all variables in the outermost procedure.

Figure 8. Referencing Variables in Nested Procedures

Storage Class DECLARE Statement Attributes

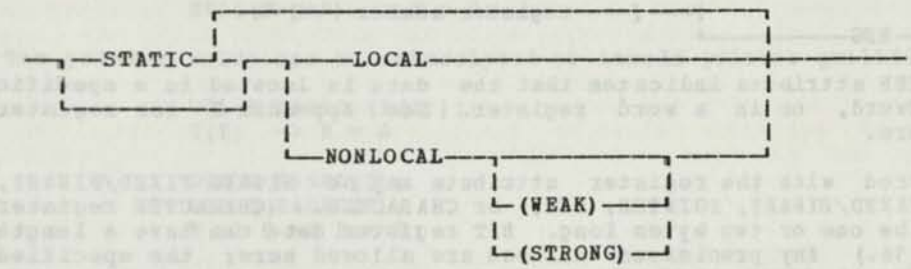
The storage class attribute tells the compiler where or how storage is to be allocated for a variable. Depending on the attribute, the compiler assigns the data to a fixed data area, to a dynamic data area, or to no data area.

If you do not specify a storage class attribute for a variable, the default is one of the following:

- STATIC NONLOCAL, if the variable is EXTERNAL and not initialized.
- STATIC LOCAL, if the variable is INTERNAL in a nonreentrant environment, or if the variable is initialized.
- AUTOMATIC, if the variable is INTERNAL and is not initialized in a reentrant environment.

STATIC Storage Class Attribute -- Fixed Storage

If you want storage for a variable assigned and never reassigned, code the keyword STATIC.



The LOCAL attribute tells the compiler to assign storage in the same area (CSECT) as the generated code. STATIC LOCAL variables can be initialized when they are declared. See "The INITIAL Attribute," later in this section, for initializing a STATIC LOCAL variable.

The NONLOCAL attribute tells the compiler not to assign storage for this variable in the PROCEDURE where this DECLARE appears. Storage is assigned where this variable is declared LOCAL.

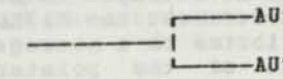
Default: A variable declared as STATIC defaults to NONLOCAL if it is EXTERNAL and not initialized. A variable declared as STATIC defaults to LOCAL if it is INTERNAL or is initialized. If only LOCAL or NONLOCAL is coded, STATIC is implied.

The STRONG attribute indicates that the compiler should generate a strong external reference (EXTRN) or a V-type address constant for the nonlocal variable. This attribute need not be specified, since it is the default.

The WEAK attribute indicates that the compiler should generate a weak external reference (WXTRN) for the nonlocal variable.

AUTOMATIC Storage Class Attribute -- Dynamic Storage

In a reentrant environment, you can cause storage to be assigned to a variable when the procedure is entered for execution. To do this, you code the keyword:



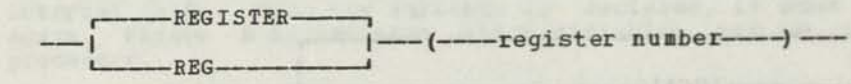
Upon exit from the procedure, the assigned storage is automatically freed.

Reentrant Environment: In order to specify the AUTOMATIC attribute, the outermost PROCEDURE statement must include the option REENTRANT. (See, "PROCEDURE -- Primary Entry Point," for details of procedure options.) Any variable declared with the AUTOMATIC attribute in an internal procedure is assigned storage at the same time as AUTOMATIC data in the external procedure.

Default: In a reentrant environment, if no storage class attribute is specified the default is AUTOMATIC if the variable is INTERNAL and not initialized.

REGISTER Storage Class Attribute

If your program assigns data to a register, you must first give the register a name and declare the name with the storage class attribute:



The REGISTER attribute indicates that the data is located in a specific byte, halfword, or in a word register. See Appendix I for register nomenclature.

Data declared with the register attribute may be SIGNED FIXED/BINARY, UNSIGNED FIXED/BINARY, POINTER, BIT, or CHARACTER. (CHARACTER register data must be one or two bytes long. BIT register data can have a length of up to 16.) Any precisions allowed are allowed here; the specified precision limits the allowable range of values for the register. The BOUNDARY and INITIAL attributes cannot be specified with the REGISTER attribute. If a precision greater than one byte is specified, use an even word register number. The variable name assigned to the register cannot be subscripted or substringed.

Registers: The register number you specify with the REGISTER attribute must take into account the register nomenclature described in Appendix I of this publication. You may specify any of the register numbers, but certain registers have compiler-assigned functions and you may not want to use these registers. The normal compiler-assigned functions for these registers depend upon whether the LINKAGE(1), LINKAGE(2), or LINKAGE(3) option of the PROCEDURE statement is in effect. See the descriptions of these options under the PROCEDURE statement description.

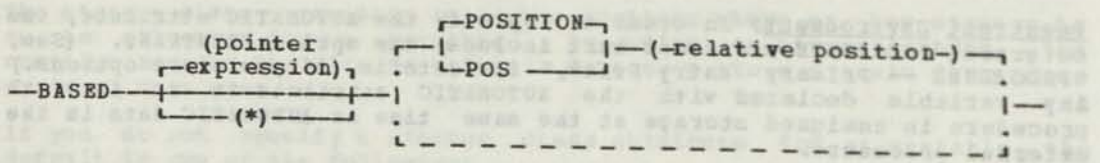
See the DECLARE attribute, RESTRICTED, for reserving registers for your own code.

Restriction on Register Declarations: "DECLARE ... REGISTER ..." statements are invalid if they appear as part of the main compiler input stream (SYSIN). They are valid only when (a) inserted with %INCLUDE or @INCLUDE statements, or (b) when obtained from SYSLIB by invoking a cataloged macro that issues an ANSWER statement that produces the register declaration in answer text.

BASED Storage Class Attribute -- Indirect Addressing

The BASED attribute allows you to use one variable name to reference different storage areas. The BASED attribute causes no storage to be assigned for the variable. Instead, whenever you reference the BASED variable the compiler applies the declared data attributes to a storage area. The storage area is located by the value of the pointer expression.

The general form of the BASED attribute is:



Using a Pointer Expression in the BASED Attribute

The pointer expression tells the compiler the location of the storage area. The compiler evaluates the expression and uses its value to locate the storage area. When you reference the BASED variable, the data attributes declared for the BASED variable are applied to this storage area.

The pointer value can be a built-in function with a pointer value such as this:

```
DCL B BASED(ADDR(A));
ADDR(A) -> B = 0;

DCL B BASED(EVAL(P-Q+R));
EVAL(P-Q+R) -> B = 0;
```

The pointer value can be subscripted or itself pointer qualified:

```
DCL M BASED (T(I));
T(I)  -> M = 0

DCL B BASED(Q);
DCL Q BASED(P);
P -> Q -> B = 0;
```

When an asterisk or no pointer expression follows BASED, a pointer value must be furnished by a RESPECIFY statement or by explicit pointer notation.

POSITION: The POSITION attribute indicates an adjustment to the value of the pointer expression. If POSITION is specified, the compiler adjusts the address to the specified relative position. For example: POS (1) is the location pointed to; no adjustment is made. POS (2) adds one bit or byte, POS (3) adds two bits or bytes, etc. If the BASED item is a bit string, the POSITION is the relative bit, and the position specification must be a multiple of eight plus one. The position specification for a bit string can be 1, 9, 17, and other such values. In all other cases it is the relative byte. The default POSITION value is 1.

If you do not include a pointer expression following the BASED attribute in the declaration, or if you write BASED(*), you must do one of two things:

1. Each time the BASED variable is referenced, provide a pointer value with the BASED variable. You would code it as pointer value -> based variable. This pointer value gives the location of the storage area.

2. Include a RESPECIFY statement somewhere before you reference the BASED variable. On the RESPECIFY statement, indicate what pointer expression is to be used to locate the storage area. (See, "RESPECIFY -- Change Register or Pointer Attributes of a Variable," for details of how to code the statement.)

Note for Structures: For structures, the BASED attribute must be specified with only the outermost structure. This causes all components of the structure to be BASED.

Providing a New Pointer Variable

Even though you declare a variable as BASED on some expression you can cause the compiler to use a different expression. There are two ways to override the expression specified in the DECLARE:

1. When you reference the BASED variable, use pointer notation.

2. Before you reference the BASED variable, include a RESPECIFY statement that provides a new default pointer expression.

Using Pointer Notation: Pointer notation consists of a pointer value followed by the composite symbol -> followed by a BASED variable. The pointer value is used to locate the storage area. The value is a pointer variable, which may be itself subscripted or pointer qualified, or a built-in function whose value is POINTER. The POSITION specified when the BASED variable is declared is not overriden by explicit pointer notation. The relative position is used with the new pointer value.

When pointer notation is used with a component of a BASED structure, the pointer is treated as if it were pointing to the beginning of the outermost structure. The location of the component is determined by combining its offset from the outermost structure with the pointer value. This does not change the value of the pointer.

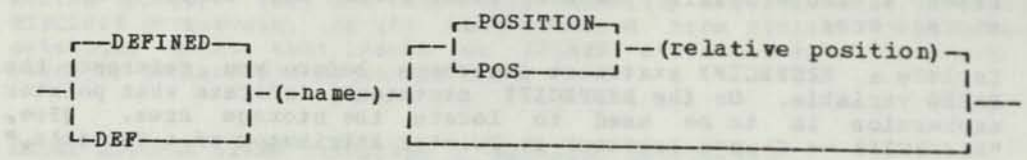
Using a RESPECIFY Statement: A pointer expression specified on the RESPECIFY statement is used to locate the BASED variable specified on that statement. The pointer expression is also used to locate any variable declared DEFINED on the BASED variable. If a POSITION was specified when the BASED variable was declared, the constant displacement is used with the new pointer variable. The BASED variable on a RESPECIFY statement cannot be the component of a structure. For details on how to code this statement, refer to "The RESPECIFY Statement."

DEFINED Storage Class Attribute -- Mapping into Another Variable

The DEFINED attribute allows you to use different variable names to reference the same storage area. You code the DEFINED attribute and the name of another variable to make the variable you are declaring represent the same storage area as that other variable. The item being declared is mapped on the item following DEFINED. You may have items DEFINED on DEFINED items; the limit is 15 levels. POSITION may be indicated. If it is, the address of the storage area is adjusted by the specified position; for example, POSITION(1) indicates no adjustment and POSITION(2) identifies the next bit or byte.

If the defining item is a bit string, the POSITION must specify a byte boundary when added to the bit offset of the defining item. For example, if the defining bit string is on a byte boundary, offset zero, the position specification can be 1, 9, 17 and other such numbers. If the defining bit string has an offset of 3, the position specification can be 6, 14, 22 and so forth.

The form of the DEFINED attribute is:



Example of DEFINED Use

It is possible to use a dummy CHAR(0) item to overlay control block definitions:

```
DCL 1 S BASED BDY(HWORD),
      2 * CHAR(2),
      2 A1 CHAR(0),
        3 B FIXED(16),
        3 C FIXED(16),
      2 A2 CHAR(0),
        3 D FIXED(16),
        3 E FIXED(16),
        3 F FIXED(16);
```

Caution: This technique should usually be avoided for the following reasons:

1. In the above example, LENGTH(S) is 2 (the sum of the lengths of the level 2 items), which is probably not the value desired.
2. A diagnostic (level 0) will be produced for each CHAR(0) item indicating that the sum of the lengths of the elements exceeds the length of the containing structure.

A better solution is the following:

```
DCL 1 S BASED BDY(HWORD),
      2 * CHAR(2),
      2 OVERLAID CHAR(MAX(LENGTH(A1),LENGTH(A2)));

DCL 1 A1 DEF(OVERLAID),
      3 B FIXED(16),
      3 C FIXED(16),

DCL 1 A2 DEF(OVERLAID),
      3 D FIXED(16),
      3 E FIXED(16),
      3 F FIXED(16);

/* LENGTH(S) IS 8 */
```

Parameter Storage Class Attribute

Input parameters for a procedure belong to the parameter storage class. There is no attribute keyword to indicate this storage class; a variable in a parameter list is automatically assigned to this storage class. References to parameters are indirect references through a list of pointers to the corresponding arguments.

CONSTANT Storage Class Attribute -- Unvarying Quantity

The CONSTANT attribute indicates an unvarying arithmetic or string quantity. It can be specified for all data types except LABEL and ENTRY data. The function of CONSTANT is to provide parameterization by providing a name for frequently used quantities.

Declaring a name with the CONSTANT attribute does not cause storage to be reserved with the given value or name. It instructs the compile phase of the compiler that it is to replace all occurrences of the declared name in source statements with the constant value before proceeding to generate any code or data for those source statements. The declared name does not appear as a symbol anywhere in the subsequent assembly phase.

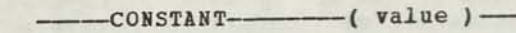
An item which is arithmetic and CONSTANT may be used as:

- A precision for FIXED data.
- A replication factor with the INITIAL attribute.
- A length for BIT or CHARACTER data.
- A dimension for arrays.
- A position for the POSITION attribute.
- An initial value.
- An item in an arithmetic expression.

An item which is string and CONSTANT may be used as:

- An initial value.
- An item in a string expression.

The general form of the CONSTANT attribute is:



The data type attributes of the value must match the data attributes declared or implied for the variable name declared as CONSTANT. If the variable is a character string, the length of the value must be the same as that declared for the variable.

Some examples of declaration and use of the CONSTANT attribute are:

```
DCL A FIXED(15) CONSTANT(2048);
DCL P PTR;
P = P + A;

DCL B FIXED(15) CONSTANT (19);

DCL CS CHAR(3) CONSTANT('ABC');
DCL VS CHAR(3) INIT(CS);

DCL BS BIT(8) CONSTANT('10110101'B);

DCL HS BIT(8) CONSTANT('C0'X);
```

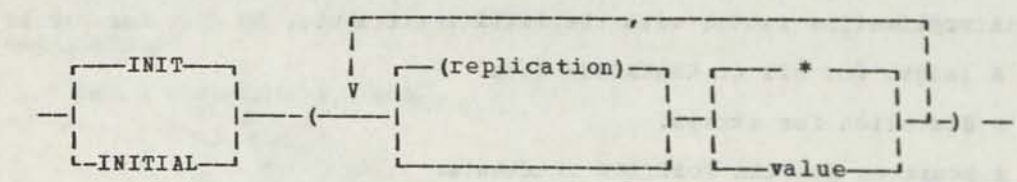
When the data type attribute of the constant is FIXED, the value can be an arithmetic expression. The expression can use the operations +, -, *, /, and //. The terms can be decimal constants, names declared as FIXED constants, or the MAX, MIN, LENGTH, or DIM built-in functions. It must be possible to evaluate the expression at compile time.

Example:

```
DCL A CHAR(13),
    B FIXED CONSTANT (2),
    C FIXED CONSTANT (11),
    D(6) CHAR(B),
    E CHAR(KKK),
    KKK FIXED
     CONSTANT(MAX(LENGTH(A), B*DIM(D), C)); /* KKK=13*/
```

The INITIAL Attribute

When you declare arithmetic, string, or pointer data that is STATIC LOCAL, you can assign it an initial value using the INITIAL attribute. The general format of this attribute is:



Replication: For arrays only, the initial value may be preceded by a replication number in parentheses to indicate that the value is to be repeated the specified number of times; ((5)'A') is equivalent to ('A','A','A','A','A'). Similarly, replication of more than the first level of the structure can be coded. A replication value may be specified as:

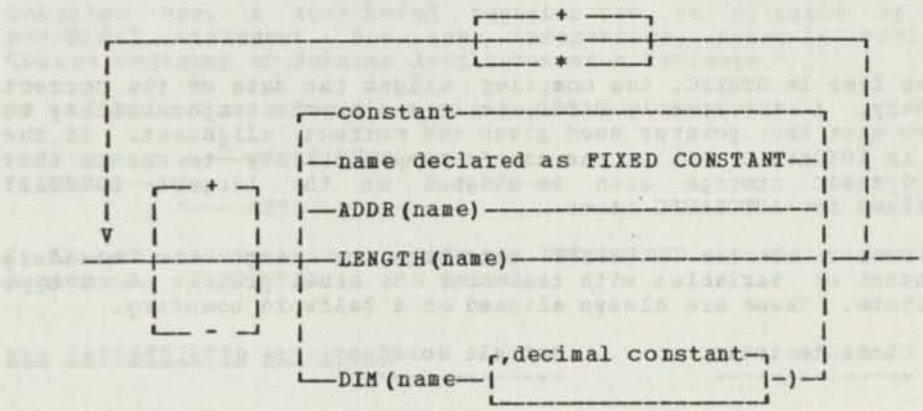
- A decimal constant.
- A binary constant.
- A name declared as a arithmetic constant.
- The LENGTH built-in function.
- The DIM built-in function.

The Initializing Value

You code an asterisk to indicate that the element should not be initialized.
You code the initializing value so that it is valid as an assignment to the declared variable; the declared variable acts as the receiver and the initializing value acts as the source expression. Refer to the discussion of the assignment statement for rules applying to different initialization techniques. This includes:

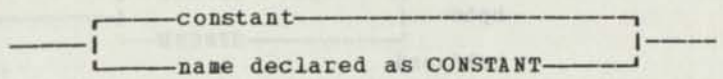
- Initializing arithmetic data with string or address data.
- Initializing string data with arithmetic or address data.
- Initializing string data with a value longer than the declared item.

If the initializing value is arithmetic, the format is:



Note: The ADDR function can only be used in an INIT expression if the name is declared as having the storage class attribute, STATIC.

If the initializing value is a character or bit string, the format is:



Initializing an Array

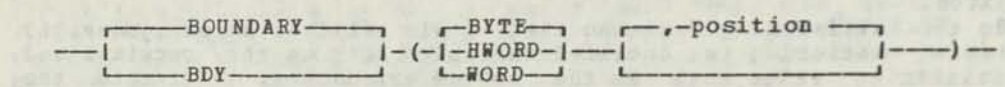
Multiple initial values separated by commas are used for array initialization. If not enough values are specified, the remaining array elements are not initialized; if too many values are specified, the remaining values are ignored. For multiple dimensioned arrays, initial values are associated according to the mapping order. For example, the array A(3,2) is mapped as: A(1,1) A(1,2) A(2,1) A(2,2) A(3,1) A(3,2). The subscripts are separately incremented in order from right to left. The maximum value of any subscript is the number of elements in that level of the array. The sequence of the values specified after the INITIAL keyword should correspond to this mapping order.

The BOUNDARY Attribute

The BOUNDARY attribute specifies the boundary alignment you desire for a variable. Do not specify the BOUNDARY attribute with the attributes REGISTER, CONSTANT, LABEL, or ENTRY.

Default: If the BOUNDARY attribute is not specified, the default alignment for the variable depends on its characteristics.

The format of this attribute is:



Position: If you specify a position, it must be a decimal number and it indicates the starting byte position within the specified boundary.

The default value for position is 1.

Arrays: Every element in an array has the same boundary and position. Components in a structure must have the same boundary or a lesser boundary than that of the containing structure. The starting byte position for each component need not be the same.

Notes:

1. If the item is STATIC, the compiler aligns the data on the correct boundary. If the item is BASED, it is the user's responsibility to ensure that the pointer used gives the correct alignment. If the item is AUTOMATIC, it is the user's responsibility to ensure that the dynamic storage area is aligned on the largest BOUNDARY specified for AUTOMATIC items.

2. You cannot use the BOUNDARY attribute to change the boundary alignment of variables with the entry or label program data type attribute. These are always aligned on a halfword boundary.

Characteristic	Default Boundary
BIT	byte (not in a structure) bit (in a structure; no dimension attribute) byte (in a structure; dimension attribute)
CHARACTER	byte
FIXED (8)	byte
FIXED(15)	halfword
FIXED(16)	halfword
FIXED(32)	word
POINTER	word
LABEL	halfword (may not be specified)
ENTRY	halfword (may not be specified)
STRUCTURE WITH NO DATA TYPE DECLARED	largest boundary required by the immediate components
STRUCTURE WITH DATA TYPE DECLARED	default boundary for the data type

The RESTRICTED Attribute

The RESTRICTED attribute may be specified with the storage class attribute, REGISTER. The RESTRICTED attribute prevents the compiler from modifying the specified register in generated code. At the start

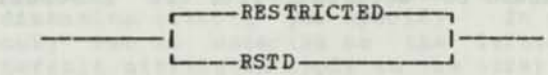
of each procedure, external and internal, the compiler assumes that all registers RESTRICTED under any name known to that procedure are not available for use in compiler generated code.

Each register variable that is declared RESTRICTED in a procedure becomes unrestricted at the end of that procedure. The specified register is restricted from the beginning of the procedure. (A register specified RESTRICTED in the program's main procedure is restricted from start to end of the program, unless a RESPECIFY statement unrestricts it.)

Note: Register variables that are restricted or unrestricted with a RESPECIFY statement remain so throughout the program regardless of the nesting level of the procedure they were declared in and subsequently respecified in. This remains in effect until they are changed with the RESPECIFY statement.

Once you no longer need a register, you should release it for use by the compiler. For a register known by more than one variable name, each name must be indicated for release before the register is free for compiler use. A restricted register can be released by using the RESPECIFY statement. For more information, refer to "RESPECIFY -- Change Register or Pointer Attributes of a Variable."

To indicate that a register is restricted you specify:

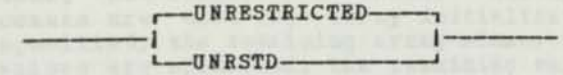


It is not necessary to specify the RESTRICTED attribute on the DECLARE statement; RESTRICTED is the default.

The UNRESTRICTED Attribute

The UNRESTRICTED attribute can be used to indicate that the register specified in the REGISTER attribute on the DECLARE statement can be used in the compiler generated code if it is needed and it if is not RESTRICTED under any other known name.

To indicate that a register is unrestricted specify:



If you want to change the restrictedness, use the RESPECIFY statement. For more information, refer to "RESPECIFY -- Change Register or Pointer Attributes of a Variable."

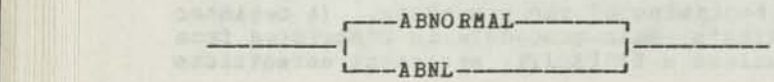
The NORMAL Attribute

The NORMAL attribute can be used to indicate that the compiler can keep a history of the variable. Keeping the history of a variable allows the compiler to produce more efficient code. It is not necessary to specify the NORMAL attribute on the DECLARE statement, since this is the default.

To specify this attribute, code:

The ABNORMAL Attribute

The ABNORMAL attribute prevents the compiler from keeping a history of a variable. To specify this attribute, code:



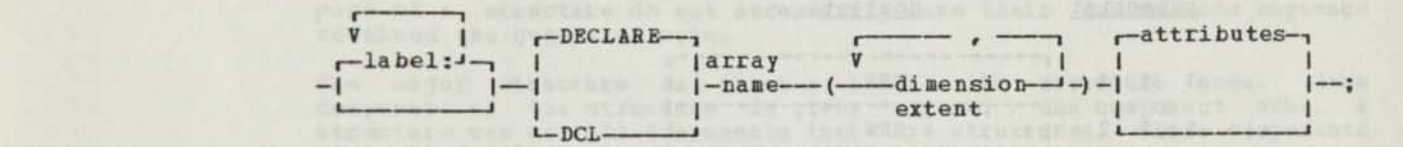
Keeping history of a variable allows the compiler to produce more efficient code. However, when variables overlap one another, and this overlapping is unknown to the compiler, incorrect history may be kept. Overlapping that is unknown to the compiler may exist with EXTERNAL variables, BASED variables, and parameters. For example:

```
DCL Q POINTER;
DCL D BASED(Q);
DCL C INIT(1);
Q = ADDR(C);      /* causes C and D to overlap                    */
V = C;            /* V is set to the value of C, which is 1       */
D = 2;            /* D, and thus C, is set to 2.                  */
V = C;            /* V would be incorrectly set to 1 (the
                     old value of C) if history were kept         */
```

In this case, the variable C should be declared with the ABNORMAL attribute.

Note: There are two other forms of the DECLARE statement. See "DECLARE -- Name a Variable and Assign its Attributes," and "DECLARE -- Specifying a Structure."

An array is a collection of data elements that have identical attributes. The array occupies a contiguous area of storage and is known by a common name. An individual element is referenced by giving its location in the array. To declare an array, code:



Dimension of an Array: Arrays can have up to 15 dimensions. Each dimension of an array has an associated extent, or number of repetitions. The total number of elements in the array is the product of all the dimension extents (for a one-dimensional array, the extent is the number of elements). If the extent is unknown and the array is not LOCAL or AUTOMATIC, you may specify an asterisk(*), as a dimension extent. For multidimensional arrays, you use commas to delimit the dimension extents you specify. In a multidimensional array, you may only use an asterisk as the leftmost dimension extent. The normal default attributes apply to the array if you do not declare a data type, scope, storage class, or BOUNDARY attribute.

Boundary

Each element in an array is aligned on the same boundary. Each element of a bit array is aligned by default on a byte boundary.

Initializing

When you declare a STATIC LOCAL array, you can initialize its elements using the INITIAL attribute. Multiple initial values separated by commas are used for array initialization. If not enough values are specified, the remaining array elements are not initialized; if too many values are specified, the remaining values are ignored. You may use an asterisk in place of an initial value to indicate that the element should not be initialized. An initial value may be preceded by a replication number in parentheses to indicate that the value is to be repeated the specified number of times. See the example below.

Referencing an Array Element

When you want to reference an element of an array, follow the array name with a subscript expression in parentheses, i.e., arrayname (subscript expression). The subscript expression gives the position of the element within the array. For multiple dimensions there must be as many expressions (separated by commas) as there are dimensions.

Subscript expressions may be any expression valid arithmetically.

Example of an Array Declaration

In the following example, EXAMP is a three dimensional array. There are two elements in each dimension. Each element is a character string with a length of 3.

```
DCL EXAMP(2,2,2) CHAR(3) INIT('EFB','NEW',(2)*,(4)'JCA');
```

The resulting elements, and their contents, is:

element subscript	content
1 1 1	EFB
1 1 2	NEW
1 2 1	<------- undefined
1 2 2	<------- undefined
2 1 1	JCA
2 1 2	JCA
2 2 1	JCA
2 2 2	JCA

The subscripts below could be used to reference elements of this array. The associated element in the array is indicated where no variable appears in a subscript.

(1,1,2) refers to the element initialized NEW.

(1,2,2) refers to the second non-initialized element.

(2,2,2) refers to the last element in the array: JCA.

(3+J,F,T)

(B-BC,I+J,10*K)

DECLARE -- Specifying a Structure

> Note: There are two other forms of the DECLARE statement. See "DECLARE -- Name of Variable and Assign its Attributes," and "DECLARE -- Specifying an Array."

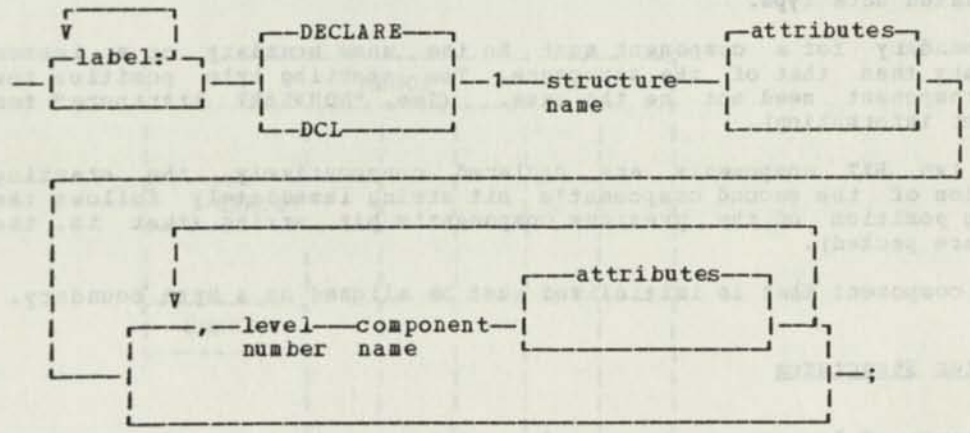
A structure is a collection of data that includes scalar data, arrays, and structures. This collection of data usually has unlike attributes; however, the data can have identical attributes.

If data items must have a specific sequence in the compiled code, declare them in that sequence and as a part of a structure. Items not a part of a structure do not necessarily have their source code sequence retained the generated code.

The major structure is given a name: the structure name. Each component of the structure is given a name: the component name. A structure can contain components that are structures. These components are called substructures or minor structures.

When you reference the structure name, you are referring to the entire collection of data. When you reference a component name, you are referring only to the component.

To declare a structure, code:



Level Numbers: Level numbers indicate the hierarchy of the components in a structure. The major structure name always has a level number of 1, which indicates that this name is at the highest level. Component names may be at the second level or lower. Each component name must have a level number greater than its containing structure, and, if it in turn is a structure, it must be followed immediately by the component names that make up its structure. The level number of a component must be equal to or less than the level number of the immediately preceding component at the same level. The following is a structure of components in a single DECLARE statement:

```
DCL 1 STRUCT,
      3 FULLWD FIXED(32),
        5 ITEMA FIXED(15),
        5 ITEMB CHAR(2),
      3 HALFWDS FIXED(15),
        5 ITEMC FIXED(8),
        5 ITEMD BIT(8);
```

If the component, "3 HALFWDS," were coded "4 HALFWORDS," it would not follow the "immediately preceding" rule just given; it would become a substructure of "3 FULLWD", and not a component separate from that structure.

Attributes: The attributes described for declaring a variable apply to structures as well.

Attributes can be assigned to the major structure name and to any of the declared component names. If you do not declare a data type attribute for the major structure or for a substructure, the default data type is CHARACTER, with a length sufficient to span all of its components. If you do not declare a data type for any component name that is not a structure, the default is FIXED(15).

You can only declare a storage class attribute with the major structure name. All components are in the same storage class as the structure. Also, you can only declare the NORMAL or ABNORMAL attribute with the major structure name.

You can use the INITIAL attribute to initialize an entire structure or any component of a structure.

Boundary

If you do not explicitly declare the data type or boundary of a structure, the structure is aligned on the strongest boundary required by immediate components. If you declare the data type but not the boundary, the structure is aligned on the default boundary for the associated data type.

The boundary for a component must be the same boundary or a lesser boundary than that of the structure. The starting byte position for each component need not be the same. (See, "BOUNDARY Attribute" for further information).

When two BIT components are declared consecutively, the starting position of the second component's bit string immediately follows the ending position of the previous component's bit string (that is, the bits are packed).

A BIT component that is initialized must be aligned on a byte boundary.

Register Structures

Structures of length one, two, or four bytes can have the storage class attribute REGISTER. The declaration must be from a SYSLIB, as in other register declares. (See, "Restriction on Register Declarations," in the description of the REGISTER attribute of the DECLARE statement.)

One byte structures can be associated with any of the 32 byte positions in the lower halfword portion of the registers.

Two byte structures can be associated with any of the 16 lower halfword positions of the registers.

Four byte structures can be associated with any of the 16 even-numbered registers. The high-order two bytes will be held in the upper halfword of the register and the low-order two bytes will be held in the lower halfword of the register. There are several restrictions on the declaration and use of four byte register structures:

- The major structures can only be POINTER or FIXED(32).
- Any component that is four bytes long can only be POINTER or FIXED(32).
- Any component that is not four bytes long cannot span the boundary between the left and right halves of the structure.
- No component of less than two bytes can be in the upper half of the major structure.

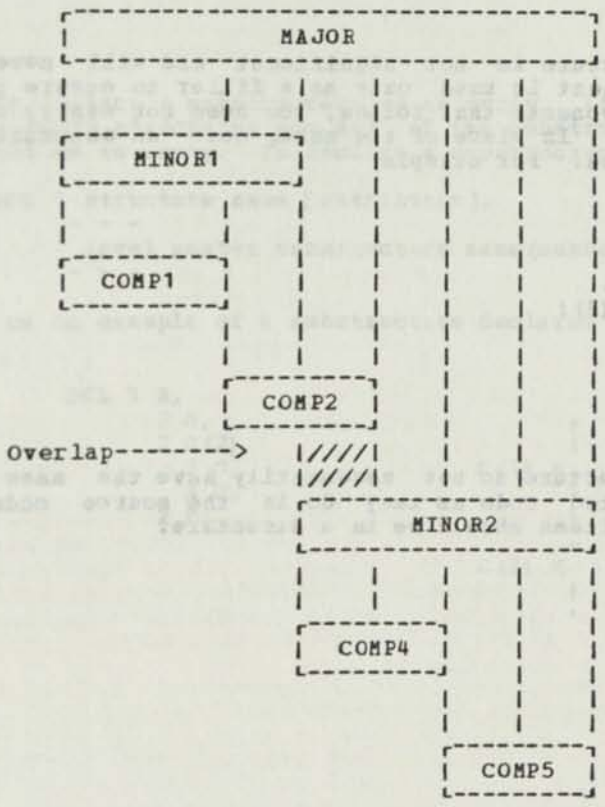
- Substring expressions cannot refer to part of the lower half of the major structure.
- Substring expressions cannot span the boundary between the upper and lower halves of the major structure.

No register structure, nor its components, can have either the DIMENSION attribute or the BOUNDARY attribute.

Overlapping

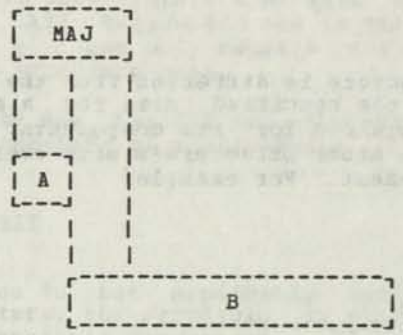
You can specify that the size of a structure is different from the size required to span its components. If the specified size for a minor structure is smaller than the size required for its components, the portion of data that extends beyond the minor structure's size overlaps with the data of the next declared component. For example:

```
DCL 1 MAJOR,
      2 MINOR1 CHAR(3),
        3 COMP1 CHAR(2),
        3 COMP2 CHAR(2),
      2 MINOR2 CHAR(4),
        3 COMP4 CHAR(2),
        3 COMP5 CHAR(2);
```



Note: When the specified size of the major structure is less than the size of all of its components, the next data item declared after the structure does not overlap with the structure. For example,

```
DCL 1 MAJ CHAR(4),
        2 A CHAR(2),
        2 B CHAR(8),
DCL C CHAR(1);
```



Name Placeholders

If the name of a structure is not significant and will never be referenced, or if a component is used only as a filler to ensure proper boundary alignment of components that follow, you need not supply a name on the DECLARE statement. In place of the name, code an asterisk; the compiler will supply a name. For example:

```
DECLARE 1 *,
          2 ITEM1,
          2 ITEM2,
          2 * CHAR(3),
          2 BYTE CHAR(1);
```

Ordering Data Items

Data items not in a structure do not necessarily have the same order (sequence) in the generated code as they do in the source code. If ordering is required the items should be in a structure.

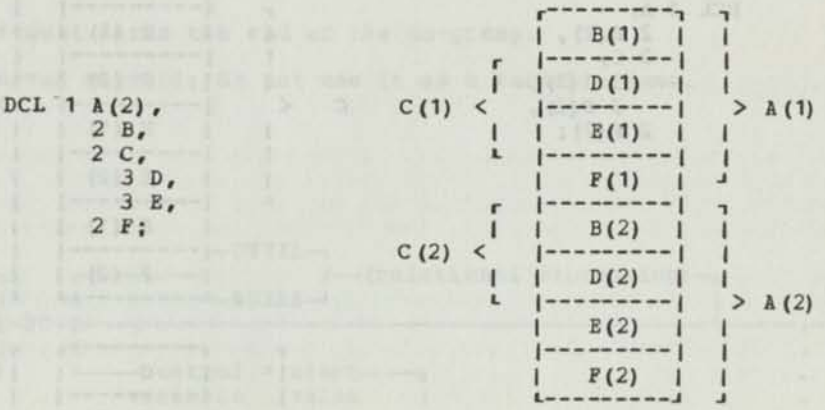
Arrays of Structures

You may declare the outermost structure as an array. If you do this, no component name can be declared as an array. To declare the outermost structure as an array, code:

```
DCL 1 structure name(dimensions) [attributes]

      [,level number component name [attributes]]...;
```

Below is an example of an outermost structure declared as an array:

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

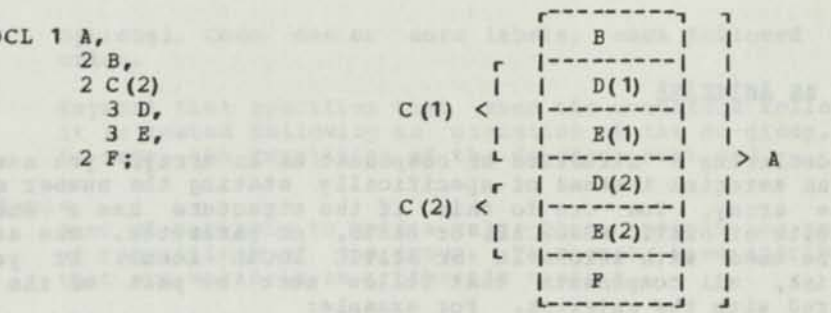


You may declare a substructure as an array. If you do this, neither any of the outer structures nor any of the substructure's components can be declared as an array. To declare a substructure as an array, code:

```
DCL 1 structure name [attributes],
    - - -
      level number substructure name(number of elements)[attributes],
    - - -
```

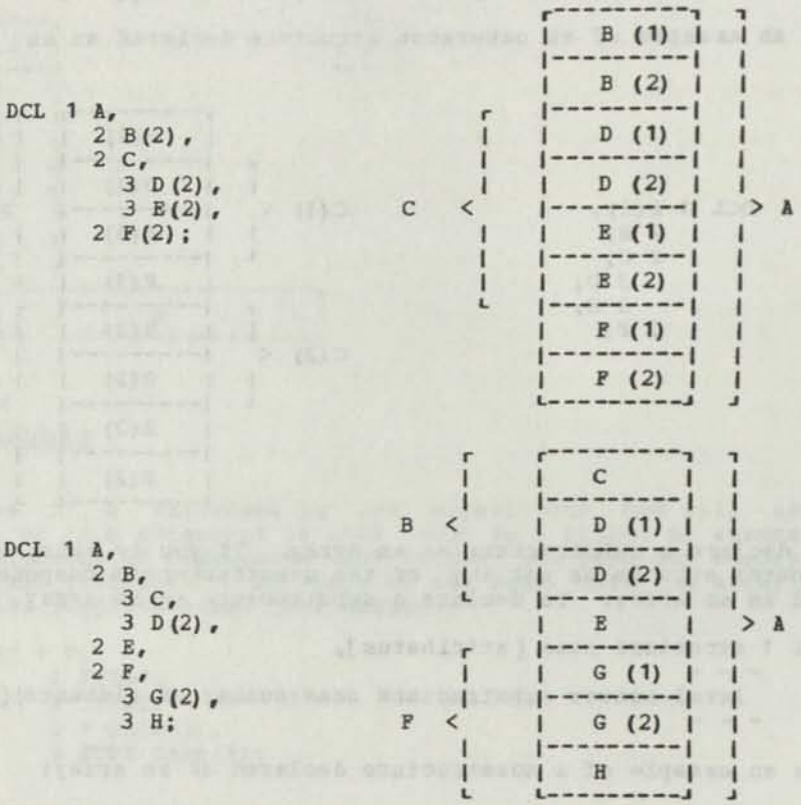
Below is an example of a substructure declared as an array:

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



You may declare any component that does not describe a structure as an array. If you do this, none of the structures of which this component is a part can be declared as arrays. To declare a component as an array, code:

```
DCL 1 structure name [attributes],
    - - -
     level number component name(number of elements)[attributes],
    - - -
```



Using an Asterisk

When declaring a structure or component as an array, you sometimes can code an asterisk instead of specifically stating the number of elements in the array. You can do this if the structure has a storage class attribute of STATIC NONLOCAL, or BASED, or parameter. The asterisk may not be used with AUTOMATIC or STATIC LOCAL items. If you code an asterisk, all components that follow must be part of the component declared with the asterisk. For example:

```
DCL 1 A,  /* ASSUME A IS A PARAMETER */
    2 B,
    2 C,
    2 D(*),
      3 E,
      3 F;
```

Note: You can also use an asterisk as a string length. See, "String Data Type DECLARE Statement Attributes," in the descriptions of DECLARE statement attributes.

DO -- Start of Do-Group

Purpose

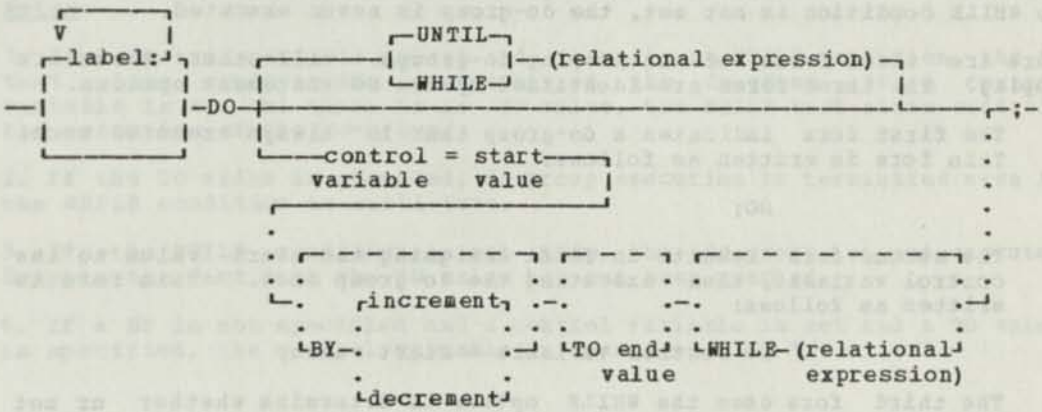
Marks the first of a do-group: a set of statements that the compiler will build to execute once or a number of times, based on what you specify in the DO statement.

Rules

The END statement marks the end of the do-group.

DO is a reserved keyword; do not use it as a variable name.

Syntax



Operands

label — Optional. Code one or more labels, each followed by a colon.

UNTIL — Keyword that specifies that when the condition following it is tested following an execution of the do-group, and is true, the repetition of the do-group must end.

control variable — Name of variable to hold a value that controls the number of repetitions of the group. This must have attributes that are valid in an arithmetic context.

start value — A constant, variable, or expression whose value is to be placed into the control variable.

BY — Keyword that specifies that the control variable is to be bumped up or down at each iteration of the do-group.

increment — Specifies how much to add to the control variable before comparing it to the end value.

decrement — Specifies how much to subtract from the control variable before comparing it to the end value. The first non-blank character after the keyword BY must be a minus sign. The minus sign indicates that the BY value is a decrement instead of an increment.

TO — Keyword that specifies that an end value is to be compared to the control variable after each execution of the do-group.

end value — A constant, variable, or expression that specifies the incremented/decremented value of the control variable at which repetition of the do-group is to end.

WHILE — Keyword that specifies that the condition following it must be true for repetition of the do-group to occur.

relational expression — The condition following the WHILE or UNTIL keyword must be any relational expression valid for an IF statement.

Non-Looping Do-Groups

A non-looping do-group describes a do-group that is executed once. Termination normally occurs after the single execution of the group of statements. The WHILE condition may be used to control execution; if the WHILE condition is not met, the do-group is never executed.

There are three forms of non-looping do-groups. All other forms are looping. The three forms are identified by the DO statement options.

1. The first form indicates a do-group that is always executed once. This form is written as follows:

```
DO;
```

2. The second form results in first assigning the start value to the control variable, then executing the do-group once. This form is written as follows:

```
DO control variable = start value;
```

3. The third form uses the WHILE option to determine whether or not the do-group is executed. If the WHILE condition is met, this form results in one execution of the do-group. If the WHILE condition is not met, the do-group is not executed. This form is written as follows:

```
DO control variable = start value
    WHILE (   expression    );
```

Looping Do-Groups

A looping do-group describes a do-group that might be executed more than once in succession or that might not be executed at all.

There are several forms of the DO statement that cause looping do-groups. The different forms correspond to the combinations of options specified in the DO statement (refer to the syntax diagram of the DO statement).

To provide control of the looping, certain information should be included with the options. The value specified with TO and the condition specified with WHILE or UNTIL control termination of the loop. If TO is specified, there must be control variable initialization and there may be a BY value.

How Options Control Looping Do-Groups

When you code a control variable, Figure 9 shows how the information you specify with the options is used in controlling execution of looping do-groups. Steps 2 and 3 provide mechanisms for exit from the loop. Steps 1, 2 and 4 are components that use the control variable. Each step is associated with one of the options:

OPTION	STEP
Start value assignment	Step 1.
The TO option	Step 2.
The WHILE option	Step 3.
The BY option	Step 4.

Figure 10 shows how the WHILE option controls execution of a looping do-group with no control variable specified.

Figure 11 shows how the UNTIL option controls execution of a looping do-group with no control variable specified.

Notes:

1. If a control variable is set and there is no WHILE condition, the TO test alone controls the execution of the do-group. If a control variable is set and there is no TO value, the WHILE test alone controls the execution of the do-group.

2. If the TO value is exceeded, do-group execution is terminated even if the WHILE condition is still true.

3. If the WHILE condition is not true, the do-group is not executed despite the fact that the TO value has not been reached.

4. If a BY is not specified and a control variable is set and a TO value is specified, the control variable is incremented by 1.

The Control Variable

The control variable must be an arithmetic variable, or a string variable of one or two bytes. The control variable can be referenced within the do-group. If you modify the control variable within the do-group, the modified value is used in determining whether or not to execute the do-group again.

If only the control variable initialization is specified on the DO statement, the start value is assigned to the control variable and the group of statements is executed once. If a WHILE expression is also specified with the control variable setting, the control variable is initialized and the group of statements may be executed once, provided the WHILE condition is met.

On exit from the do-group, the control variable is the value at the termination of the loop. If no looping occurred, the control variable equals the start value assigned to it.

The Start Value

The start value can be any constant, variable, or complex expression that is valid as an arithmetic value. (See, "Assignment -- Assign a Value to a Variable," for a detailed explanation of expressions.)

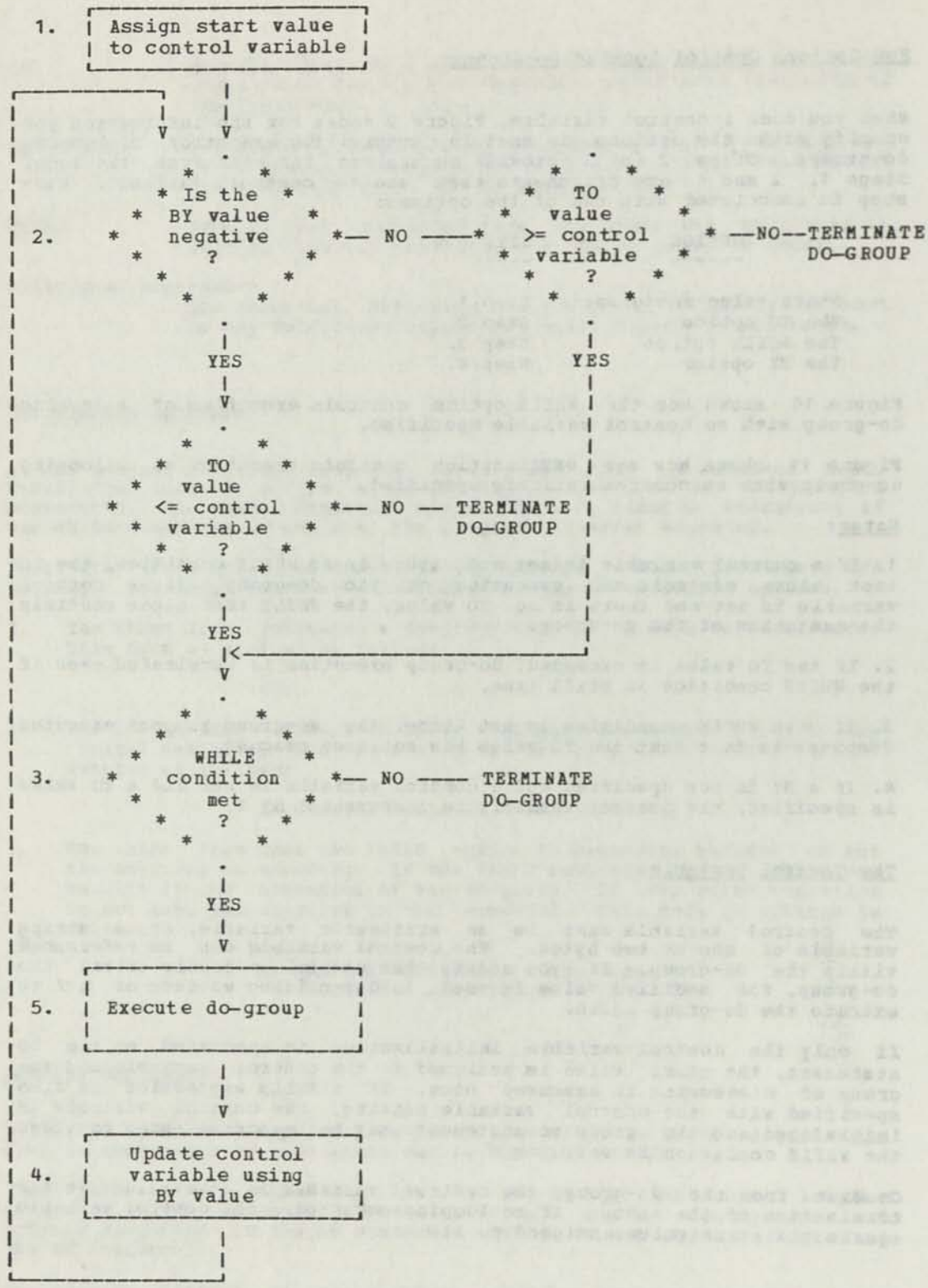


Figure 9. Do-Group Execution -- Control Variable Specified

The BY Value

The BY value can be any constant, variable or complex expression that can be used in an arithmetic expression. If a BY value is not specified, but the start value and the TO value are specified, the BY value defaults to 1.

The BY value is recalculated on each iteration of the loop. Therefore, if it is not a constant, the BY value can change on each iteration.

If the first non-blank character after the BY keyword is ((or is not)) a minus sign, the BY value is subtracted from ((or added to)) the control variable on each iteration of the loop. The resulting control variable value is then compared to the TO value to determine whether or not the loop terminates. The loop terminates if the control variable is less than ((or greater than)) the TO value.

The TO Value

The TO value can be any constant, variable, or complex expression valid when used arithmetically.

The TO value will be recalculated on each iteration of the loop. Therefore, if it is not a constant, the TO value can change on each iteration of the loop.

If you do not code a TO value, but you code a BY value, the loop is potentially infinite; some exit mechanism must be included in the do-group. The WHILE option may be used to provide exit control.

The WHILE Value

The WHILE value can be any relational expression. If the result of the expression is true, the do-group is executed. If the result is false, the do-group is not executed.

If both a TO value and a WHILE expression are coded, the TO value comparison is made before the WHILE expression is evaluated.

The UNTIL Value

The UNTIL value can be any relational expression. If the result of the expression is false, execution of the do-group continues. If the result is true, execution of the do-group ends.

If an UNTIL expression is coded, then it is not permissible to code a control variable, start value, BY value, TO value, or WHILE expression.

Nested Do-Groups

You can include up to 14 nesting levels of DO statements within a do-group. A DO statement and its associated statements within a do-group are executed the specified number of times each time the do-group is encountered. For example:

```
LOOP10:   DO I = 1 TO 10;      /*EXECUTE LOOP 10 TIMES          */
LOOP5:      DO J = 1 TO 5;     /*EXECUTE LOOP 5 TIMES           */
              A(I,J) = X;      /*CONTROL VARIABLE IS SUBSCRIPT*/
            END;
          END;
```

...LOOP10 is executed 10 times. LOOP5 is executed 5 times for every single execution of LOOP10. In total, LOOP5 is executed 50 times.

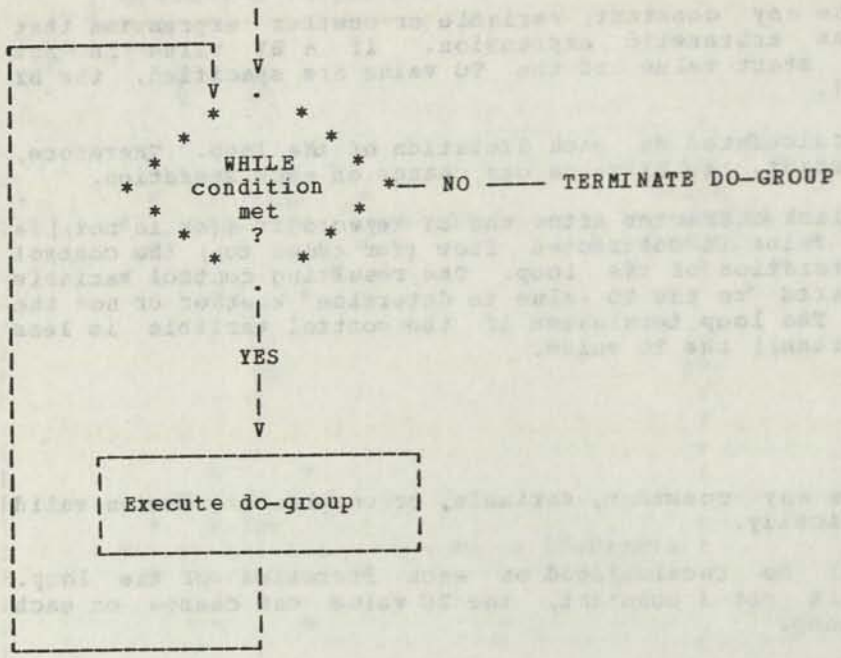


Figure 10. Do-Group Execution -- WHILE Option Only

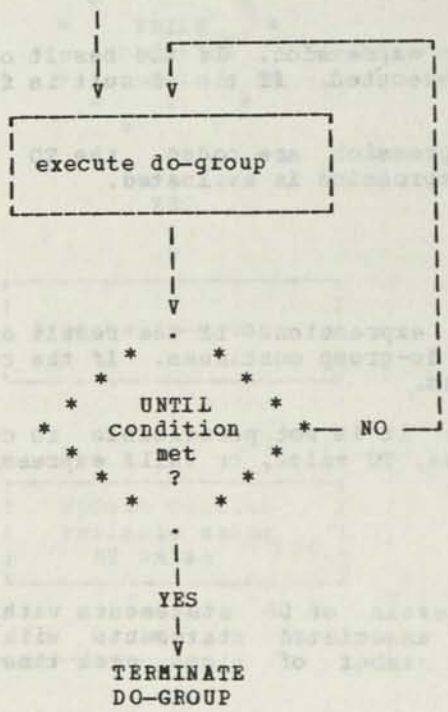


Figure 11. Do-Group Execution -- UNTIL Option Only

END -- End of Procedure or Do-Group

Purpose

Tells the compiler that this is the end of the statements for the present procedure or do-group.

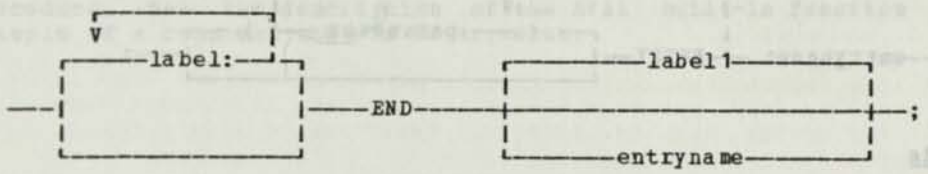
Rules

For an END statement at the end of a procedure, the compiler generates a return to the calling procedure.

For an END statement at the end of a looping do-group, the compiler generates a test of the conditions for looping that you specified in the DO statement.

END is a reserved keyword; do not use it as a variable name.

Syntax



Operands

label — Optional. Code one or more labels, each followed by a colon.

entryname — The name, on the PROCEDURE statement, of the procedure that this END statement applies to.

label1 — The label of the DO statement introducing the do-group that this END statement applies to.

ENTRY -- Secondary Entry Point

Purpose

Designates a secondary entry point for a procedure. (The primary entry point is the procedure's PROCEDURE statement.)

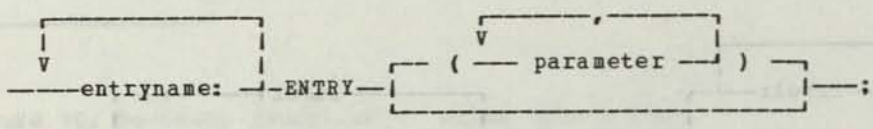
Rules

The number of secondary entry points in a procedure is not restricted.

Use the CALL statement to invoke a procedure at a secondary entry point. You may pass arguments with the CALL by coding their associated parameters on the ENTRY statement.

ENTRY is a reserved keyword; do not use it as a variable name.

Syntax



Operands

entryname Code at least one to specify the name you will use in the CALL statement for this secondary entry point.

parameter Code the names of the variables with which the procedure references the arguments passed in the CALL statement.

Declaring the Entry Name

When declaring the entry name that you reference in the ENTRY statement, the DECLARE statement appears in the calling procedure. If you do not declare the entry name, appropriate defaults are assigned when the entry name is encountered. For a detailed description of how to declare an entry name, refer to the discussion of "Program Data" in the DECLARE statement section.

Specifying Entry Arguments and Parameters

Parameters on an ENTRY statement must be simple data item names, each separated by a comma. The maximum number of parameters is 16. The number and position of the parameters on a PROCEDURE and ENTRY statement need not be the same.

There is a correspondence between the parameters on the ENTRY statement and the arguments on the CALL statement that invokes this entry point. When you include arguments on a CALL statement, a parameter list is produced. Each argument is assigned one word in the parameter list; an address is inserted in each word. The parmreg points to this parameter list across the linkage.

The compiler associates CALL statement arguments with ENTRY statement parameters in the order in which they appear; the first argument is associated with the first parameter, the second argument with the second

parameter, and so on. Since the association of arguments and parameters is by order, it does not matter whether the variable names used are the same or different. However, when CALL statement arguments are passed to an internal procedure, the variable names used for the ENTRY statement arguments must be different from the names used for the parameters in the CALL statement.

Arguments are declared in the calling procedure; parameters are declared in the called procedure. The data type attributes declared for a parameter should be the same attributes declared for the corresponding argument in the calling procedure. Parameters are automatically assigned to the PARAMETER storage class; therefore, do not declare a storage class attribute for a parameter.

Modifying an Entry Parameter

Assigning a value to a parameter results in modifying the corresponding argument. This has no effect on the calling procedure unless the actual address of the argument was inserted in the parameter list or the argument was a constant. (The type of argument that has its actual address inserted in the parameter list is described in "The CALL Statement" under the topic "How the Arguments Are Passed.")

You should not modify a parameter whose corresponding argument is a constant since this could cause unpredictable results in the calling procedure. See the description of the EVAL built-in function for an example of a constant that is an argument.

MACGEN, @ENDGEN -- Assembly Statement Delimiters

Purpose

Allows you to include a subset of DPPX Assembler statements for processing by the assembly phase.

Rules

The MACGEN statement is valid only when produced as ANSWER text by a PL/DS macro definition invoked from SYSLIB.

The DPPX Assembler PL/DS "subset" limitations are: no invocations of DPPX Base assembler macros and no DPPX Assembler macro definitions may be in the assembler statements between the MACGEN and @ENDGEN statements.

Any text contained between the statements MACGEN and @ENDGEN is passed unchanged to the assembly phase and is not processed by the compile phase.

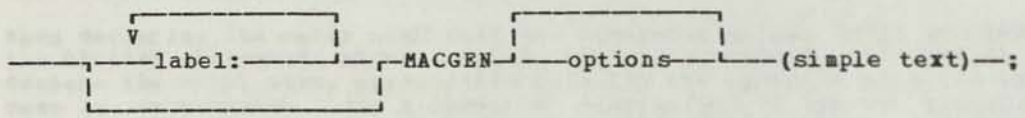
The compiler has no knowledge of the contents of the text without help from options on the MACGEN statement. For optimization purposes, you must code the options.

Without this information, incorrect code may be generated. For example, if the compiler does not know that a variable is referenced within the text, it may assign that variable to a register, causing errors in the assembly phase.

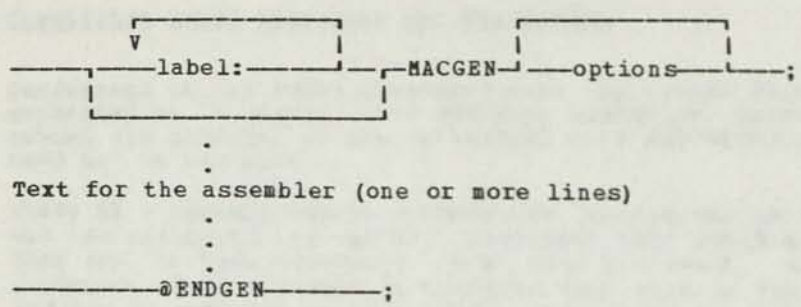
MACGEN and ENDGEN are reserved keywords; do not use them as variable names.

Syntax

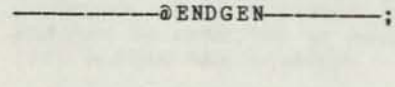
1. A single line of text may be passed to the assembly phase using the following syntax:



2. If several lines of text are required, the following syntax should be used:

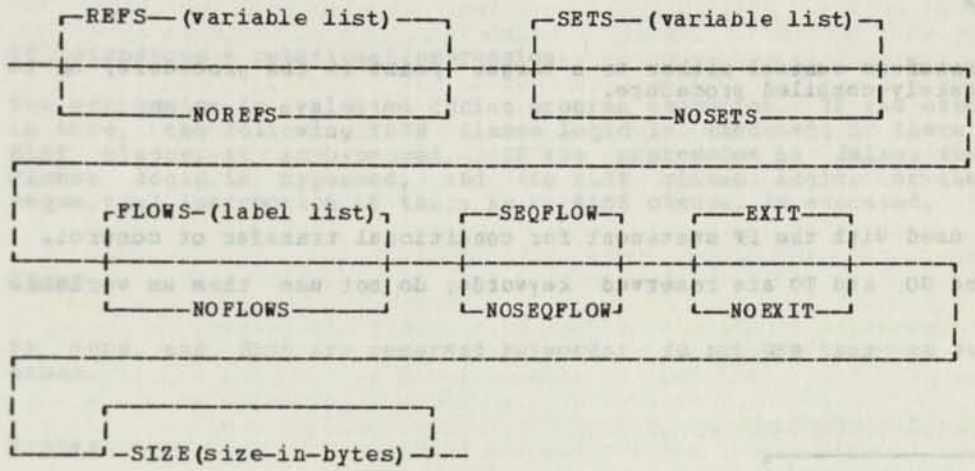


Text for the assembler (one or more lines)



Options Syntax

In both cases, the syntax of the options parameter is:



REFS(variable list)/NOREFS	List the variables referenced but not set in the assembler statements. Default: NOREFS.
SETS(variable list)/NOSETS	List the variables whose values are set in the assembler statements. Default: NOSETS.
FLOWS(label list)/NOFLOWS	List the labels in any other section of code in this compilation that the assembler statements might branch to. Default: NOFLOWS.
SEQFLOW/NOSEQFLOW	SEQFLOW specifies that the logic flow of the assembler statements may continue into the statement following @ENDGEN. Default: SEQFLOW.
EXIT/NOEXIT	EXIT specifies that the logic flow might not return at all to this assembly module. Default: NOEXIT.
SIZE(size-in-bytes)	Specify in bytes the maximum size of the code that the assembly phase will generate from these statements.

The author of the macro should ensure that the correct information is specified by means of the MACGEN options.

GOTO -- Transfer Control

Purpose

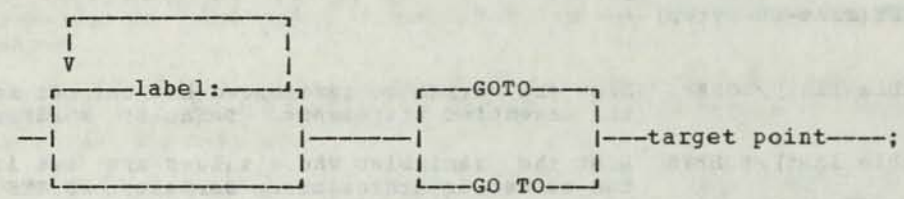
GOTO transfers control either to a target point in the procedure, or to a separately-compiled procedure.

Rules

Can be used with the IF statement for conditional transfer of control.

GOTO and GO and TO are reserved keywords; do not use them as variable names.

Syntax



Operands

label — Optional. Specify one or more labels, each followed by a colon.

target point — Code the label of the statement you want to branch to. If you use a based label, declare it with the VALUERANGE data type attribute of the DECLARE statement.

Example of a Branch Using a BASED LABEL

```
DCL LAB BASED VALUERANGE(LAB1,LAB2,LAB3) LABEL,
    LPTR(3) PTR INIT(ADDR(LAB1),ADDR(LAB2),ADDR(LAB3));

GOTO LPTR(I)->LAB;
```

Alternatively, if LAB is declared BASED(LPTR(I)), the GOTO statement can be simplified to

```
GOTO LAB;
```

IF THEN (ELSE) -- Conditional Execution

Purpose

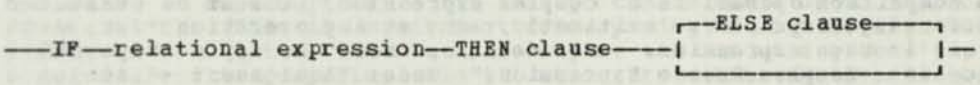
IF introduces a relational expression.

The expression is evaluated during program execution. If the expression is true, the following THEN clause logic is executed; if there is an ELSE clause, it is bypassed. If the expression is false, the THEN clause logic is bypassed, and the ELSE clause logic, or the next sequential instruction if there is no ELSE clause, is executed.

Rules

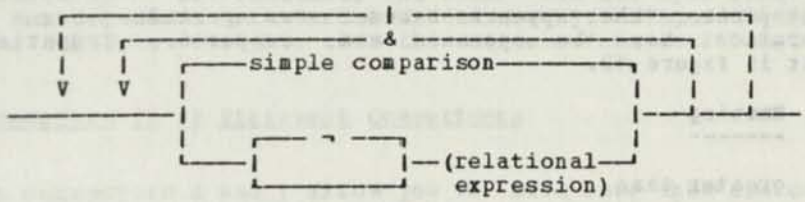
IF, THEN, and ELSE are reserved keywords; do not use them as variable names.

Syntax



Relational Expression Syntax

The form of a relational expression is:



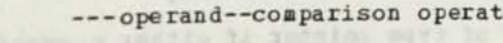
Relational Expression Operands

& The 'and' sign (&) is used as a connector of comparisons. If both comparisons are true, the result is true; otherwise, the result is false. If the left comparison is false, the right comparison is not tested. See, "Connectors in IF Statement Comparisons," below.

| The 'or' sign (|) is used as a connector of comparisons. If both comparisons are false, the result is false; otherwise, the result is true. If the left comparison is true, the right comparison is not tested. See, "Connectors in IF Statement Comparisons," below.

¬ The 'not' sign (¬) causes the negation of a relational expression. If the relational expression is true, the result is false. If the relational expression is false, the result is true.

Simple Comparison Syntax



Simple Expression IF Statement Operands

The operands on the IF statement can be arithmetic or string constants, or variables, or complex expressions. Variables may be FIXED, POINTER, CHARACTER, or BIT. If a variable describes an element of an array, the variable will include a subscript expression. If a variable describes part of a character or bit string, the variable will include a substring expression.

See the topic, "String Constants," under "Assignment -- Assign a Value to a Variable." See also, "Character-String Comparisons" and "Bit-String Comparisons," below.

Operators in an IF Statement Comparison Operand

See "Operators in the Source Expression," under "Assignment -- Assign a Value to a Variable."

Operation Performed on the Operand

When a comparison operand is a complex expression, it must be evaluated by performing a pointer, arithmetic, or string operation for each operator in the expression. For details, see the topic "Operation Sequence in a Complex Source Expression," under "Assignment -- Assign a Value to a Variable."

IF Statement Comparison Operators

The comparison operator that appears between two operands on an IF statement determines how the operands are compared. Comparison operators appear in Figure 12.

Operator	Meaning
>	greater than
<	less than
¬>	not greater than
¬<	not less than
=	equal to
¬=	not equal to
>= or =>	greater than or equal to
<= or =<	less than or equal to

Figure 12. IF Comparison Operators

Operation Performed on IF Statement Comparisons

The type of comparison made is pointer, arithmetic, or string.

Pointer Comparisons: The comparison is of type pointer if either operand is pointer data. If the other operand is not pointer data, it should be unsigned, and will be padded on the left with zeros before the comparison is made. The result is unsigned.

Arithmetic Comparisons: The comparison is arithmetic if neither of the operands is pointer data, and at least one is arithmetic. If the operand of greater precision is unsigned, the comparison is unsigned; otherwise it is signed. As a consequence of this rule, the only signed comparisons are of FIXED(15) with either FIXED(8) or FIXED(15).

Character-String Comparisons: The comparison is a character-string comparison if neither operand is FIXED or POINTER, and if one or both operands are CHARACTER. The comparison proceeds from left to right. The comparison result is based on the hexadecimal character codes of the values compared. Both operands must have the same length. The following is an exception: a character expression can be compared to a null string (''). This is equivalent to comparing the string to blanks.

> Note: A character string that is compared to a bit string has a length that is eight times its character length. String operands that are eight bit multiples aligned on a byte boundary are limited to a maximum of 256 bytes.

Bit-String Comparisons: The comparison is a bit-string comparison when both operands are of type bit (variable or constant). In this case, the only valid comparison operators are = and ¬=. Either operand can be preceded by the ¬ operator, which indicates that the ones complement is to be taken prior to performing the comparison. Both operands must have the same length and this length must be known at compile time. The following is an exception: a bit string can be compared to the null bit string (''B). This is equivalent to comparing the bit string to a bit string of zero bits. You can also compare the bit string to ¬''B, which is equivalent to comparing the bit string to a bit string of one ("1") bits.

If both operands are variables and either one of them is not byte-aligned, or one of them has a length that is not a multiple of eight, then each variable must be contained within two bytes. If one operand is a bit constant, the other can not span more than 32 bytes. If both operands are variables and are byte-aligned with a length that is a multiple of eight, then the variables can be 256 bytes in length.

Connectors in IF Statement Comparisons

The connectors & and | allow you to have more than one comparison on an IF statement. Figure 13 shows how the result of one comparison affects whether another comparison will be made. If another comparison is not made, the result of the first comparison determines the result of the connector.

The connector & has a higher priority than the connector |.

If the connector is &, the relation on the left of the & connector is tested. If the left relation is true, the relation on the right determines the truth or falsity of the &. If the left comparison is false, the & is false.

If the connector is |, the relation on the left of the | connector is tested. If the left relation is false, the right relation determines the truth or falsity of the |. If the left relation is true, the | is true.

The ¬ as a logical operation reverses truth or falsity at that point. For example, ¬(A>B) is equivalent to A¬>B.

Comparison#1 Result	Connector Character	Comparison#2 Result		Combined Result
T	&	T	=	T
T	&	F	=	F
F	&	T (not tested)	=	F
F	&	F (not tested)	=	F
T	|	T (not tested)	=	T
T	|	F (not tested)	=	T
F	|	T	=	T
F	|	F	=	F

Figure 13. Summary of Results of Connected IF Comparisons (True-False)

The THEN Clause in the IF Statement

The THEN clause immediately follows the relational expression, and it is executed only if the relation is true. The THEN clause can be one of: a null statement, a single statement (other than an END statement, an ENTRY statement, or a PROCEDURE statement), or it can be a group of statements (a do-group). A THEN clause can be another IF statement, called a nested IF statement. Nested IF statements are described after the discussion of the ELSE clause.

The ELSE Clause in the IF Statement

The ELSE clause appears after the THEN clause. It is executed only when the relation is false. The ELSE clause can be one of: a null statement, a single statement (other than an END statement or PROCEDURE statement), or it can be a group of statements (a do-group). An ELSE clause can be another IF statement, called a nested IF statement. Nested IF statements are described below.

Note: The ELSE keyword cannot be preceded by labels. Labels are allowed after the ELSE keyword on the statement(s) of the ELSE clause.

Nested IF Statements

When an IF statement appears in a THEN or ELSE clause, it is called a nested IF statement. A nested IF statement is coded exactly like any IF statement. The maximum level of nesting for IF statements is 14.

Within a nest of IF statements, the innermost ELSE clause is associated with the innermost THEN clause; the next innermost ELSE clause is associated with the next innermost THEN clause, and so on. If an incorrect association would be made because an IF has a THEN clause but does not require an ELSE clause, follow the THEN clause with a null ELSE clause. (ELSE; is a null ELSE clause.)

In the following example if the inner null ELSE were omitted, the second ELSE would be associated with the second rather than the first IF.

```
X = 0;
IF A = B THEN
  IF I>J THEN
    X = 1;
  ELSE;
ELSE
  X = 2;
```

Null Statement -- No Compiler Action

Purpose

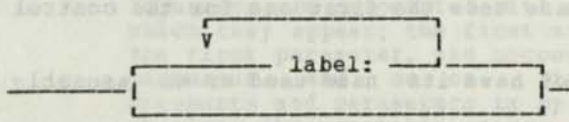
You use a null statement to indicate that you want no compiler action at this point in your program.

The null statement is useful for special situations in conditional (IF) constructions. You would use it after the keyword THEN when no action is to be taken if the relational operator is true. You would use it after the keyword ELSE when it is necessary to provide proper association of THEN and ELSE clauses in a nest of IF statements.

Rules

When a null statement follows the THEN Keyword, and the IF comparison is true, then the compiler-generated code's execution continues with the instruction following the IF statement.

Syntax



Operands

label Optional. Code one or more labels, each followed by a colon.

Purpose

The PROCEDURE statement indicates the primary entry point for a procedure. (The ENTRY statement is used to indicate secondary entry points.)

Rules

A PROCEDURE statement always appears as the first statement of each external or internal procedure, including the main procedure. However, a PROCEDURE statement for an external procedure can be preceded by a MACGEN statement or RESPECIFY statements that restrict registers.

Options on the PROCEDURE statement cause the PL/DS generated code for the statement to be altered.

The LINKAGE(3) option of the PROCEDURE statement provides enhanced intra-modular linkage for execution under DPPX Base.

The name that you code on the PROCEDURE statement of the main procedure becomes the name of the control section (CSECT) that the assembly phase builds from that procedure. If you code several names for the PROCEDURE statement, the assembly phase uses the first one for the control section name.

A nested procedure does not have its name used as an assembly control section name.

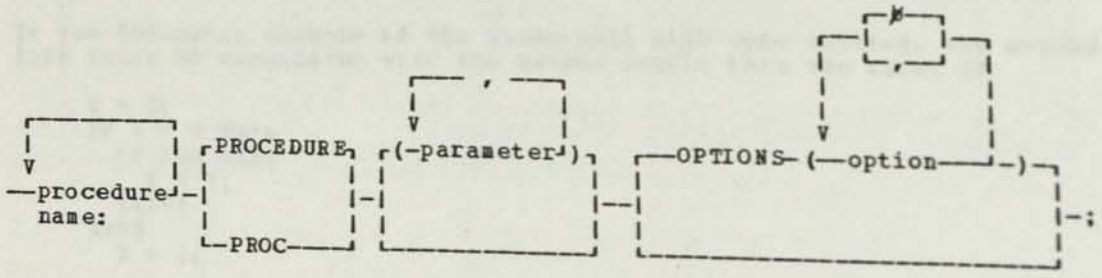
A PROCEDURE statement and an END statement begin and end a procedure.

An external procedure can contain up to 254 procedures, called internal procedures. Internal procedures can contain procedures up to a nesting level of 14. An internal procedure can be invoked only by its containing procedure or by any other procedure within the containing procedure.

Data declared in an external procedure is known to all its internal procedures. Data declared in an internal procedure is known only to itself and its internal procedures. An internal procedure may not declare a variable with the same name that was declared in a containing procedure or another internal procedure.

PROC and PROCEDURE are reserved keywords; do not use them as variable names.

Syntax



Operands

procedure name — The procedure name is the name you want assigned as the primary entry point name. The first name will be the name of the CSECT. You may specify multiple procedure names, each followed by a colon.

In the calling procedure, you should declare the name of a called procedure as ENTRY. If you do not declare the procedure name, appropriate defaults are assigned when the procedure name is first encountered. For a detailed description of how to declare a PROCEDURE name, refer to the discussion of Program Data in the DECLARE statement section.

parameter — The parameter is a variable name that is associated with an argument. Parameters on a PROCEDURE statement must be simple data names, each separated by a comma. The maximum number of parameters is 16.

There is a correspondence between the parameters on the PROCEDURE statement and the arguments on the CALL statement that invokes this procedure. When you include arguments on a CALL statement, a parameter list is produced. Each argument is assigned one word in the parameter list; an address is inserted in each word. The parameter register points to this parameter list.

Arguments are associated with parameters in the order in which they appear; the first argument is associated with the first parameter, the second argument with the second parameter, and so on. Since the association of arguments and parameters is by order, it does not matter whether the variable names used are the same or different, when passing parameters between separately compiled procedures. However, when arguments are passed to an internal procedure, the variable names used for the arguments must be different from the names used for the parameters.

Arguments are declared in the calling procedure; parameters are declared in the called procedure. The data type attributes declared for a parameter should be the same attributes declared for the corresponding argument in the calling procedure. Parameters are automatically assigned to the parameter storage class; therefore, do not declare a storage class attribute for a parameter.

Assigning a value to a parameter results in modifying the corresponding argument. This has no effect on the calling procedure unless (a) the actual address of the argument was inserted in the parameter list or (b) the argument was a constant. (Which arguments have their actual address inserted in the parameter list is described in the description, "Call -- Invoke an Internal or External Procedure," under the topic "How the Arguments Are Passed.") You should not modify a parameter whose corresponding argument is a constant since this could cause unpredictable results in the calling procedure.

option — You must specify OPTIONS and the options you have selected only in the PROCEDURE statement of your program's main procedure.

The options are keywords that are used to alter the standard entry and exit code provided by the PL/DS Compiler. You may specify one or more options, each separated by a comma or a blank.

You can code blanks to separate options specified in the parentheses.

There are options available to alter compiler-generated entry and exit code. These options and their associated keywords are shown under "Summary of PROCEDURE Statement Options."

Note: A number of PROCEDURE options described below permit you to specify registers for various purposes. You should ensure that the register numbers you specify do not conflict with the register conventions associated with the LINKAGE option you choose.

Summary of PROCEDURE Statement Options

MAIN: Indicates that no module calls this module and that no registers are saved on entry or restored on exit.

REENTRANT: Indicates that compiler-generated code should be reentrant.

LINKAGE: Specifies the type of linkage code to be generated by the compiler at prologs and epilogs. (Certain values of LINKAGE allow procedure options not listed in this table. See Appendix E.)

ID: Specifies that an identifying character string is to appear at the front of the procedure being compiled.

NOID: Specifies that no identifying character string is to appear at the start of the procedure being compiled.

ENDID: Specifies that an identifying character string is to appear at the end of the procedure being compiled.

NOENDID: Specifies that an identifying character string is not to appear at the end of the procedure being compiled.

SAVE: Specifies the registers to be saved on entry to and restored on exit from the procedure being compiled.

NOSAVE: Specifies the registers that are not to be saved on entry and restored on return from the procedure being compiled.

AUTODATA: Indicates the maximum permissible size of the dynamic DSECT that forms the AUTOMATIC data of a reentrant procedure.

NOAUTODATA: Indicates that there should be no dynamic DSECT even though the procedure is reentrant.

AUTOREG: Specifies the register to be used to address AUTOMATIC data.

NOAUTOREG: Specifies that the compiler should not set up any register to address AUTOMATIC data.

TEMPS: Indicates that compiler-generated temporary storage areas need not be avoided.

NOTEMPS: Indicates that, where possible, the compiler should avoid generating temporary storage areas.

STATREG: Specifies the register(s) to be used to address STATIC data.

NOSTATREG: Specifies that the compiler should not set up any register to address STATIC data.

SAVEAREA: Indicates that the compiler is to create a save area for the use of CALLed procedures.

NOSAVEAREA: Indicates that the compiler is not to create a save area for the use of CALLed procedures.

SAVEREG: Specifies the register to be used to locate save areas to hold copies of registers at inter-module interfaces.

NOSAVEREG: Indicates that no register is to be used to locate save areas.

RETREG: Specifies the register to be used for the BALR instruction in CALL statements and for the matching BR instruction in RETURN statements.

NORETREG: Indicates that there is no return register.

RTOREG: Specifies the register to be used to hold the address of a branch target in a RETURN-TO statement.

NORTOREG: Indicates that no register is to be used to hold the address of a branch target in a RETURN-TO statement.

PARMREG: Specifies the register to be used to locate parameter lists at CALL, PROCEDURE, or ENTRY statements.

NOPARMREG: Indicates that no register is to be used to locate parameter lists at CALL, PROCEDURE, or ENTRY statements.

RCODREG: Specifies the register to be used to pass a return code value back to a CALLing procedure.

NORCODREG: Indicates that no register is to be used to pass a return-code value back to a CALLing procedure.

BRANREG: Specifies the register to be used to hold the address of CALLed entry points.

NOBRANREG: Indicates that no register is to be used to hold the address of CALLed entry points.

WORKREGS: Specifies the registers to be avoided by the global register assignment phase.

NOWORKREGS: Indicates that the global register assignment phase can use all non-restricted registers.

MAIN -- Procedure Statement Option

The MAIN option is coded as:

```
--------------------MAIN----------------------
```

MAIN specifies that no registers are saved on initial entry to the procedure, and that a RETURN or the corresponding procedure END statement corresponds to either: (a) a loop stop of the form "J *", for LINKAGE(1) or LINKAGE(2); or (b) a DPPX "EXIT" function for LINKAGE(3).

Parameters may be passed to a MAIN procedure. It is the user's responsibility to ensure that PARMREG points to an appropriate parameter address list.

If MAIN is specified, the options SAVE and NOSAVE are invalid.

REENTRANT -- Procedure Statement Option

The REENTRANT option is coded as:

```
-------------------REENTRANT----------------------
```

When REENTRANT is specified as an option on a PROCEDURE statement, the compiler generates reentrant code. The external procedure and its internal procedures are reentrant as a unit; internal procedures are not

separately reentrant. Storage for AUTOMATIC data in the external and internal procedures is obtained as one block at the beginning of the procedure, and freed at the end.

REENTRANT has an effect on procedure option defaults and declaration defaults. If REENTRANT is specified, the default storage class for declarations is AUTOMATIC, and the procedure options AUTOREG(28) and NOSTATREG are assumed by default.

If REENTRANT is not specified, and if it is not implied by specifying one or more of the options AUTOREG, AUTODATA, NOAUTOREG, or NOAUTODATA, then the default storage class for data is STATIC LOCAL.

LINKAGE -- Procedure Statement Option

The LINKAGE option is coded as:

```
------------LINKAGE-----(---number---)-------
```

This option indicates which linkage conventions should be observed by the compiler.

LINKAGE(0): Indicates that the compiler should not generate any prologue or epilogue code at all. No CSECT card will be generated, but the end of the generated assembler code for the module will be marked by an assembler END statement. It is the user's responsibility to ensure that any dynamic storage required is obtained and that any required or defaulted AUTOREGs or STATREGs are loaded with correct values.

LINKAGE(1): Indicates that the compiler should generate a CSECT card for the main name of the procedure, and ENTRY cards for any secondary names, but should not generate any code to save or restore registers, or to provide addressability to STATIC or AUTOMATIC data. It is again the user's responsibility to ensure that any dynamic storage required is obtained, and that registers are loaded with correct values.

See Figure 14 for the register conventions that are in effect for LINKAGE(1).

LINKAGE(2): Is the default value. The compiler saves and restores registers, and provides addressability to data areas.

See Figure 14 for the register conventions that are in effect for LINKAGE(2).

LINKAGE(3): Simplifies intra-modular linkage in code for execution under DPPX Base. It allows several additional procedure options, such as: GETAUTO, USESTACK, SUBPOOL, and EID.

The LINKAGE(3) option imposes its own set of register conventions. See Appendix E for details.

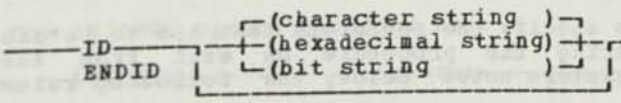
Address Register	Function
0	Used for prologue and epilogue code. Not restored after use.
16	Points to a parameter list across the linkage if the PROCEDURE statement contains parameters or if you call a procedure and pass arguments.
2	Used to hold the return code 'n' if RETURN CODE(n) is specified.
28 or other register(s) chosen by user or never restricted by user	Used as a base register for addressing STATIC data. (No register is used unless specified by user). See STATREG procedure option.
	Used as a base register for addressing AUTOMATIC data in a reentrant environment. See AUTOREG procedure option.
22	Used by the standard save mechanism. Register 22 points to the save area.
24	Used across linkage as a "return to" location pointer when control is returned to an invoking procedure.
10	Used across linkage to contain the address of the entry point of any procedure that is called. Not restored after use.
All other registers, and above at times when they are not in use	Used when necessary for arithmetic calculations, indexing, external data addressing, and pointer manipulation.
Procedure options are described with the PROCEDURE statement.	

Figure 14. LINKAGE(1) or LINKAGE(2) Register Conventions

ID, ENDID, and NOID -- Procedure Statement Options

The ID and ENDID options produce for a module an identifying character string, hexadecimal string, or bit string. The ID Option places the string in the object code at the front of the module. The ENDID places the string at the end of the module. Either or both options may be coded.

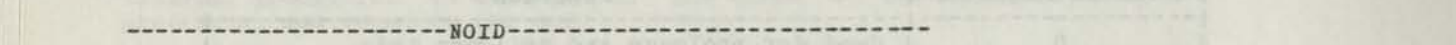
The ID and ENDID options are coded as:



The character string you specify with the option can be up to 256 characters long. If one of the characters is an apostrophe, code two consecutive apostrophes in its place.

If you do not specify a character string following the keyword ID, the compiler generates a name based on the first (or only) name for the procedure.

The NOID Option is coded as:

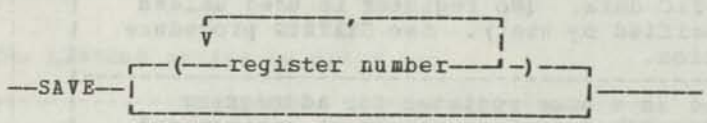


The NOID option indicates that no identifying character string is to be generated.

If neither option is coded, the default is ID, followed by no string.

SAVE -- Procedure Statement Option

The SAVE option is coded as:



The SAVE option allows you to specify the registers that are saved at the start of the procedure and restored on exit from the procedure. (For optimization, all registers are saved on entry to the procedure.) Except for the registers noted below, the following rules apply:

(a) If SAVE is specified with no operands, all registers are restored.

(b) If SAVE is specified with operands, the registers specified are saved and are restored on exit from the procedure, but the registers that are not specified retain whatever contents they have at the end of the procedure.

The default is SAVE with no operands. Do not code SAVE and NOSAVE in the same procedure.

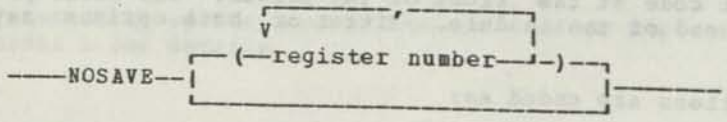
Notes

1. With LINKAGE(3) in use, you cannot specify a register list with SAVE.

2. The use of the SAVE option requires that the caller of the routine has provided the address of his save area with the register specified with the PROCEDURE statement SAVEREG option.

NOSAVE -- Procedure Statement Option

The NOSAVE option is coded as:



The NOSAVE option allows you to specify the registers that are to retain the values they acquired during the procedure on exit from the procedure. Except for the registers noted below, the following rules apply:

(a) If NOSAVE is specified with no operands, all register contents at the end of the procedure are retained, and no registers are restored.

(b) If NOSAVE is specified with operands, the contents of the registers specified are retained and are not restored on exit from the procedure. The registers not specified are saved on entry to the procedure and their values are restored on exit from the procedure. (If the register list is omitted, the calling program need not supply a save area. If a list is supplied, the generated code requires a save area and all registers may be stored on entry.)

Do not code SAVE and NOSAVE in the same procedure.

Note: With LINKAGE(3) in use, you cannot specify a register list with NOSAVE.

Registers Eligible for SAVE and NOSAVE: Only even-numbered registers from 0-30 may be specified. Figure 15 shows the save area's format.

Word	Contents
1	Chain-back address (third word of calling procedure's save area)
2	Chain-forward address (third word of called procedure's save area)
3-6	Lower halfwords of primary registers
7-10	Upper halfwords of primary registers
11-14	Lower halfwords of secondary registers
15-18	Upper halfwords of secondary registers

Figure 15. Format of the Save Area for LINKAGE(2)

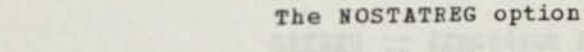
STATREG -- Procedure Statement Option

The STATREG option is coded as:

Any even register from 2 through 30, except 10, 24 and 26, may be specified as a base register. The register provides addressability to 32K bytes of storage. By default, no register will be provided, and register 0 will be used. This provides addressability to STATIC data within 32K of the current instruction. You need only use STATREG if this default addressing is inadequate. Do not use PARMREG if there are any CALL statements with parameters in the procedure.

NOSTATREG -- Procedure Statement Option

The NOSTATREG option is coded as:



If this option is coded, no register will be assigned to point to STATIC data. NOSTATREG is always the default.

AUTOREG -- Procedure Statement Option

The AUTOREG option is coded as:

Any even register from 2 through 30 can be specified as a base register. The default is 28. The register provides addressability to 32K bytes of AUTOMATIC storage. PARMREG should not be used if there are any CALL statements with arguments in the procedure.

AUTOREG is the default if REENTRANT is specified, and will force REENTRANT if AUTOREG is specified without REENTRANT. If STATREG is also coded, the registers chosen must not overlap.

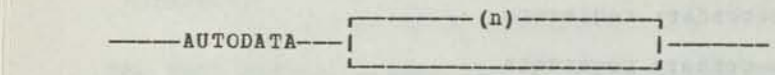
NOAUTOREG -- Procedure Statement Option

The NOAUTOREG Option is coded as:

If this option is coded, there will be no addressability to any AUTOMATIC storage required by the procedure. This option is the default if REENTRANT is not specified. If it is specified together with REENTRANT, it implies that, although the procedure is reentrant, it does not require any AUTOMATIC storage.

AUTODATA -- Procedure Statement Option

The AUTODATA option is coded as:



This option is used to control the size of the dynamic DSECT created for a REENTRANT procedure. If the size of the DSECT exceeds n bytes, an error message is produced. AUTODATA(0) means the same as NOAUTODATA, and implies that no dynamic DSECT will be required. If no value is coded, the option simply means that a DSECT is expected.

The default is AUTODATA with no value, if REENTRANT is specified, and NOAUTODATA if REENTRANT is not specified. If REENTRANT is not specified, AUTODATA forces the option REENTRANT.

NOAUTODATA -- Procedure Statement Option

The NOAUTODATA option is coded as:

This option specifies that no dynamic data area is required for this procedure. If the compiler generates any dynamic data, an assembler error message is produced. If REENTRANT is not specified, NOAUTODATA forces REENTRANT.

TEMPS -- Procedure Statement Option

The TEMPS option is coded as:

The TEMPS option specifies that the compiler is to generate temporary storage areas as required. If neither TEMPS nor NOTEMPS is coded, TEMPS is the default.

NOTEMPS -- Procedure Statement Option

The NOTEMPS option is coded as:

This option specifies that the compiler should avoid generating temporary storage areas where possible. If the compiler cannot avoid generating a temporary storage area, it issues a warning message.

NOSAVEAREA -- Procedure Statement Option

The NOSAVEAREA option is coded as:

```
------------------------NOSAVEAREA--------------------------
```

This option specifies that there should be no save area allocated for the use of called procedures.

Normally, the compiler allocates a save area if a CALL statement or a MACGEN statement appears in the program being compiled. This procedure option causes the compiler to not create such a save area.

Note: This option is frequently needed when the options REENTRANT and NOAUTODATA are coded.

SAVEREG -- Procedure Statement Option

The SAVEREG option is coded as:

```
--------------------SAVEREG(register)--------------------
```

This option specifies the register to be used to locate register save areas. The register must be an even-numbered register and must not be 0.

Note: LINKAGE(3) generates prolog and epilog code to invoke linkage assist routines that are dependent on this register number. Therefore, changing the register number means that the user must replace the linkage assist routines.

NOSAVEREG -- Procedure Statement Option

The NOSAVEREG option is coded as:

```
--------------NOSAVEREG-------------------------
```

This option specifies that no register is to be used to locate register save areas. This is invalid if either of the options SAVEAREA or SAVE are explicitly coded or supplied by default.

RETREG -- Procedure Statement Option

The RETREG option is coded as:

```
--------------RETREG(register)----------------------
```

This option specifies the register to be used to effect inter-procedure linkage. It is set by CALL statements (BALR instruction) and used by RETURN statements. The register must be an even-numbered register and must not be 0.

Note: LINKAGE(3) generates prolog and epilog code to invoke linkage assist routines that are dependent on this register number. Therefore, changing the register number means that the user must replace the linkage assist routines.

NORETREG -- Procedure Statement Option

The NORETREG Option is coded as:

```
-----------NORETREG-----------------------
```

This option specifies that no register is to be used to effect inter-procedure linkage. This is invalid if LINKAGE(3) is coded.

RTOREG -- Procedure Statement Option

The RTOREG Option is coded as:

```
--------------RTOREG(register)-----------------
```

This option specifies the register to be used to effect inter-procedure linkage. It is used by RETURN-to statements to hold the address of the return-to label. The register must be an even-numbered register and must not be 0.

For LINKAGE(3), the default is 26. For other values of LINKAGE, the default is to the register used for RETREG.

Note: LINKAGE(3) generates prolog and epilog code to invoke linkage assist routines that are dependent on this register number. Therefore, changing the register number means that the user must replace the linkage assist routines.

NORTOREG -- Procedure Statement Option

The NORTOREG option is coded as:

```
-------------NORTOREG--------------------
```

This option specifies that no register is to be used to effect RETURN-TO linkage. This is invalid if any RETURN-TO statements appear in the procedure being compiled.

RCODREG -- Procedure Statement Option

The RCODREG option is coded as:

```
-------------RCODREG(register)-----------------
```

This option specifies the register to be used by RETURN-CODE statements to return an expression value to the CALLing procedure. The register must be an even-numbered register and must not be 0.

If LINKAGE(3) is coded the default is Register 30. For other values of LINKAGE, the default is Register 2.

NORCODREG -- Procedure Statement Option

The NORCODREG option is coded as:

```
---------------NORCODREG-----------------------
```

This option specifies that no register is to be used to contain RETURN-CODE values. This is invalid if RETURN-CODE statements are in the procedure being compiled.

PARMREG -- Procedure Statement Option

The PARMREG option is coded as:

```
----------PARMREG(register)----------
```

This option specifies the register to be used to locate parameter address lists at CALL, PROCEDURE, and ENTRY statements. The register must be an even-numbered register and must be 0.

Note: LINKAGE(3) generates prolog and epilog code to invoke linkage assist routines that are dependent on this register number. Therefore, changing the register number means that the user must replace the linkage assist routines.

NOPARMREG -- Procedure Statement Option

The NOPARMREG option is coded as:

```
----------NOPARMREG----------
```

This option specifies that no register is to be used to locate parameter address lists. This is invalid if any CALL, PROCEDURE, or ENTRY has a parameter list.

BRANREG -- Procedure Statement Option

The BRANREG option is coded as:

```
----------BRANREG(register)----------
```

This option specifies the register to be used as a branch register. This register is used to hold the address of the target entry of a CALL statement. The register must be a primary even-numbered register and must not be 0.

If LINKAGE(3) is coded the default is 8. For other values of LINKAGE, the register is defaulted to 10.

Note: LINKAGE(3) generates prolog and epilog code to invoke linkage assist routines that are dependent on this register number. Therefore, changing the register number means that the user must replace the linkage assist routines.

NOBRANREG -- Procedure Statement Option

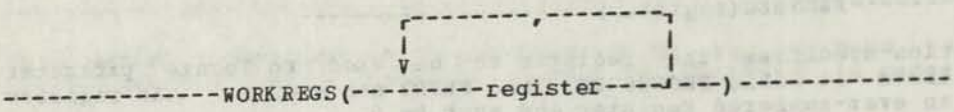
The NOBRANREG option is coded as:

```
----------NOBRANREG----------
```

This option indicates that no register is to be used as a branch register.

WORKREGS -- Procedure Statement Option

The WORKREGS option is coded as:



This option specifies one or more registers that are not to be used by the register assignment algorithm during optimization. These registers are reserved for the compile phase's final code generation phase.

This option is not normally needed. It is supplied for the use of programmers who have a stringent optimization problem.

NOWORKREGS -- Procedure Statement Option

The NOWORKREGS option is coded as:

This option indicates that the register assignment phase of optimization is to use all non-restricted registers. This option should be used with care. It can lead to more highly optimized code, but it exposes the user to the possibility of a failure-to-compile situation because code generation cannot find any registers to evaluate expressions in.

RESPECIFY -- Change Register or Pointer Attributes of a Variable

Purpose

For REGISTER variables, the RESPECIFY statement can be used to make an unrestricted register variable restricted or a restricted register variable unrestricted. For BASED variables, the RESPECIFY statement can be used to replace the present pointer value for locating the data.

Rules

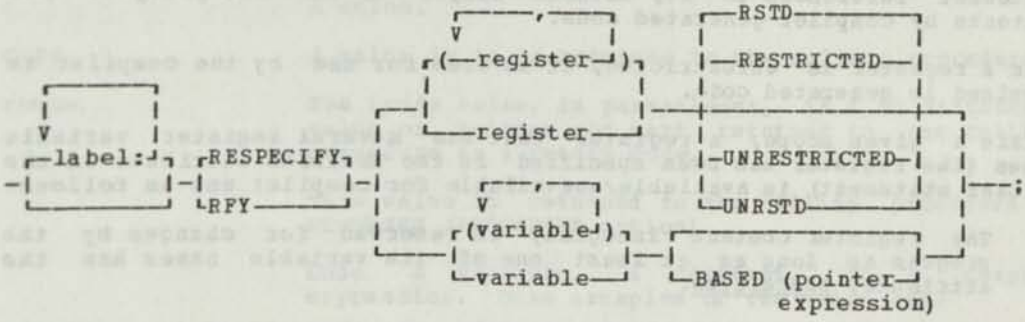
The RESPECIFY statement functions only as a control on compiler processing and does not cause any executable code to be generated; the RESPECIFY statement acts as a null statement if executed.

The compiler implements the RESPECIFY statement at the place where the statement appears; this implementation is unconditional, even if the program's logic branches past the RESPECIFY statement. (Note the similarity to the assembler USING statement.)

The condition specified in the RESPECIFY statement remains in effect throughout the program unless another RESPECIFY statement changes it.

RESPECIFY and RFY are reserved keywords; do not use them as variables or label names.

Syntax



Operands

label — You may code one or more labels, each followed by a colon.

register — Code the name(s) of a variable declared register that you want restricted or released. If you code only one register variable, you need not enclose it in parentheses.

See also, "Extent of Register Restriction," in Appendix I.

RESTRICTED/UNRESTRICTED — Specify whether you want the register that you coded to be restricted or unrestricted. See "Register Restrictedness," below. (Default: If nothing is coded after the register name, the restrictedness reverts to whichever was specified or defaulted in the declaration of the register.)

variable — The variable is the name of data that is declared as BASED. If you specify only one variable name, you need not enclose it in parentheses.

pointer expression — A pointer expression is a pointer value that is used to locate the specified variable name(s). The expression:

```
ADDR(SAM)
```

...in the statement:

```
RESPECIFY JOE BASED(ADDR(SAM));
```

...is a valid pointer expression, using the ADDR built-in function.

The pointer value is used to locate the data for every variable name included on the RESPECIFY statement, for its components if a structure, and for any variable declared DEFINED on one of these based variables. The POSITION specified when the BASED variable was declared is used with the new pointer in locating the data.

(Default: If nothing is coded after the variable name, the pointer value reverts to that specified in the declaration of the variable.)

Register Restrictedness

When a register is restricted, it is reserved for use by program statement references to it, without danger to the integrity of its contents by compiler-generated code.

When a register is unrestricted, it is free for use by the compiler as required in generated code.

Within a given scope, a register that has several register variable names (the register has been specified in the REGISTER attribute of the DECLARE statement) is available/unavailable for compiler use as follows:

- The register content integrity is reserved for changes by the program as long as at least one of its variable names has the attribute, RESTRICTED.
- The register is available for compiler use when all of its register variable names have the attribute, UNRESTRICTED.

Scope: The scope of a register comprises the outermost procedure, and its nested procedures, that declare register variable names for this register.

Restrictiveness Differences Between DECLARE and RESPECIFY: Note the following differences in register variable restrictiveness between the DECLARE and RESPECIFY statements:

- A register variable declared RESTRICTED becomes unrestricted at the end of the procedure it was declared in.
- A register variable respecified RESTRICTED or UNRESTRICTED unconditionally remains so until it is respecified otherwise. This overrides the rule of declaration immediately above.

RETURN -- Return Control Back to Calling Procedure

Purpose

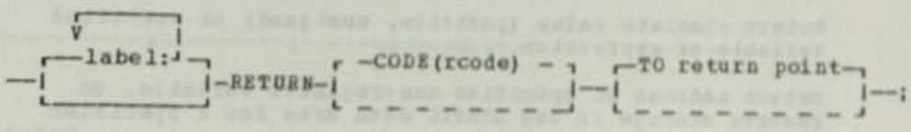
To return control from a procedure to a calling procedure before the called procedure's END statement is reached.

Rules

You can cause a return either to the statement following the CALL statement or to some other labeled statement.

A value can be returned to the calling procedure. RETURN is a reserved keyword; do not use it as a variable name.

Syntax



Operands

label — Optional. Code one or more labels, each followed by a colon.

CODE — A value is to be returned to the calling procedure.

rcode — The rcode value, in parentheses, is a an arithmetic value or pointer you want returned to the calling procedure as a return code.

This value is returned to the calling procedure in RCODEREG (PROCEDURE option).

Code a variable, a constant, or a complex expression. Some examples of the value are:

```
CODE(12)              CODE('80'X)
CODE(P -> RET)        CODE((S+4)*X)
CODE(ADDR(A))
CODE(R3 + R4)
```

TO — Code is returned to a specified return point, not to the statement after CALL.

return point — The return point is a name that identifies the statement you want to return to. It must be a label.

If you do not include the keyword TO and a return point variable on the RETURN statement, control is returned to the statement following the CALL statement in the calling procedure.

BUILT-IN FUNCTIONS FOR COMPILE STATEMENTS

The built-in functions are aids to coding compile statements. Some of them allow you to do things that cannot be done using compile instructions. Others are a compact way of performing operations that would require several statements. In most cases a built-in function is like a variable, returning a value to the place in a statement where it appears, for use by the remainder of the statement.

The keywords that identify the built-in functions are not reserved. However, you cannot use any of these keywords as both a built-in function and a variable name. If the keyword is not explicitly declared elsewhere as user data, the compiler interprets it as a built-in function.

The built-in functions are summarized in Figure 16.

Built-In Function	Use
ABS	Return absolute value (positive, unsigned) of specified variable or expression.
ADDR	Return address of specified non-register variable, or reserve storage in the STATIC data area for a specified constant.
DIM	Return extent (size) of a specified dimension in a specified array.
EVAL	Return the value of the result of the specified expression.
LENGTH	Return length of specified constant or variable.
MAX	Return the maximum value from the specified expressions.
MIN	Opposite of MAX.

Figure 16. Built-In Functions and Their Uses

ABS -- Absolute Value Built-In Function

Purpose

Returns the absolute value (positive value) of a variable or expression to the place where it is coded.

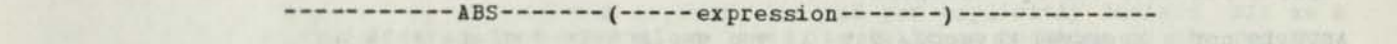
Rules

Can appear in any expression.

Its characteristics are: unsigned FIXED(16) value.

ABS is not a reserved keyword, but if you explicitly declare ABS as a variable in your program, it loses its built-in function in the program.

Syntax



Operand

expression — A variable or expression, enclosed in parentheses.

Examples of ABS Use

```
Y = ABS(T) + 10;
DO I = 1 to ABS(A-B);
X = ABS (ABS(A-B) - ABS(C/D));
IF X = ABS(C + 12) THEN GOTO FINISH;
Z = ABS(B & ST);
DO I = 1 TO ABS((A+B) * -C);
```

ADDR -- Address Built-In Function

Purpose

ADDR obtains the address of a variable or a constant. When it obtains the address of a constant, it also reserves storage to hold the value of the constant.

Rules

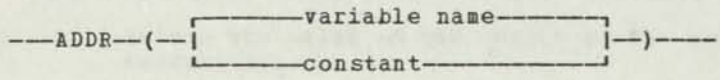
The variable named may not have the REGISTER attribute.

The variable named can be pointer qualified, subscripted, or substringed, unless ADDR appears in the value operand of the INITIAL attribute in a DECLARE statement. In this case its argument must be a simple data item name and must be static.

The variable named cannot be a bit string unless it is on a byte boundary. If it is not on a byte boundary, the compiler issues a warning and uses the byte address.

ADDR is not a reserved keyword, but if you explicitly declare ADDR as a variable in your program, it loses its built-in function in the program.

Syntax



Operands

variable name — Name the variable whose (non-register) address is required.

constant — A constant that is to have storage assigned in the static area. You should not modify this constant.

Examples of ADDR

```
DCL CARD BASED(ADDR(BUFF));          /* <-- VARIABLE */
DCL POINT PTR     INIT(ADDR(BUFF)); /* <-- VARIABLE */
X = ADDR(25);                        /* <-- VARIABLE */
```

DIM -- Array Dimension Built-In Function

Purpose

DIM obtains the extent of a dimension of an array.

Rules

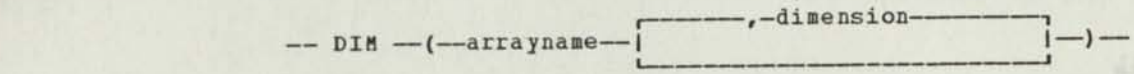
For a one-dimensional array, DIM returns the number of elements in the array.

The array name cannot be subscripted, substringed, or explicitly pointer qualified, and cannot contain operators. The DIM built-in function can appear in any position where a decimal constant is allowed, except for the level number of a structure and the second argument of DIM.

If the array is multidimensional, a dimension number may be specified, and the result of the DIM built-in function is the extent of that dimension. The value must have been declared at compile time.

DIM is not a reserved keyword, but if you explicitly declare DIM as a variable in your program, it loses its built-in function in the program.

Syntax



Operands

arrayname Name of array with the dimension whose size you are obtaining.

dimension The dimension whose extent you want. (Default: 1)

Examples of DIM

```
DCL ARX(DIM(ARRAY)) DEFINED(ARRAY);

DCL C(4) CHAR(DIM(DC7));

F = 4 * DIM(TARY);

DO I = 1 TO DIM(A);

DCL MATRIX(2,3) FIXED(15);

DCL SWITCHES BIT(DIM(MATRIX,1)*DIM(MATRIX,2));
```

EVAL -- Expression Value Built-In Function

Purpose

Provides the value, after evaluation, of the expression you specify as its argument.

Rules

It is valid to code EVAL anywhere the value of its argument has valid meaning.

EVAL is not a reserved keyword, but if you explicitly declare EVAL as a variable in your program, it loses its built-in function in the program.

Syntax

```
----------EVAL-----(---expression---)-------
```

Operands

expression The expression whose value you want used in place of the EVAL at execution time.

Uses of EVAL

Controlling Sequence of Mathematical Operations with EVAL: You can use EVAL to control the way the compiler works out the answer of an expression.

The compiler views each expression as three functional units: operator and two operands. During compiler processing, units are arranged according to the priority of the operators and their data types. You may use parentheses to group items of an expression to make the meaning of the expression more evident; however, the order of compiler evaluation of these units is not affected by parentheses.

In most cases the order of evaluation can be according to rules for operator and data type priority. However, if there is a possibility of overflow, it is important to specify the evaluation order.

Given the statement:

```
A = (B - C) + (D - E);
```

...where all variables are arithmetic and all operators are of the same priority, the compiler may perform the operations in several different orders. Some possible evaluation orders are:

```
(B+D) - (C+E)

(B-E) - (D-C)

(D+B-E) - C
```

Logically all are equivalent. However, you may want to avoid a possibility of overflow by specifying a particular order of evaluation. EVAL forces evaluation of a specified unit. Use EVAL to guarantee that the compiler will evaluate an expression in a particular order, even though there are equivalent alternatives.

In the statement:

```
A = EVAL(B - C) + (D - E);
```

...the compiler recognizes that it must evaluate (B - C) as indicated.

Temporary Variables for Changeable CALL Parameters: EVAL is useful in CALL arguments to avoid a change to a parameter affecting the argument, or to pass the value of a register variable. In the expression:

```
CALL X (EVAL(3));
```

...EVAL causes a temporary location to be assigned the value 3 and this passed as the argument. A change to the corresponding parameter will only change the temporary variable and not affect references to the constant 3.

The size of the temporary location depends on the attributes of the argument of EVAL. If the argument is a FIXED(32), a POINTER, a FIXED(32) expression, or a POINTER expression, the length is four bytes; otherwise it is two bytes.

Purpose

Obtains the length of the constant or variable that you specify.

Rules

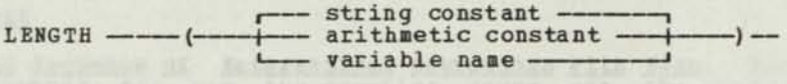
LENGTH must only be used when the length of the argument is available at compile time.

Whether the value returned by LENGTH is the number of bytes or the number of bits depends upon your specified constant's or variable's type; see Figure 17.

LENGTH can be pointer qualified, substringed, or subscripted. The LENGTH built-in function can appear in any position where a decimal constant is allowed, except for: the level number of a structure, the value in the CONSTANT attribute, and the second argument of the DIM function.

LENGTH is not a reserved keyword, but if you explicitly declare LENGTH as a variable in your program, it loses its built-in function in the program.

Syntax



Operands

string constant The string constant whose length you want.

arithmetic constant An arithmetic constant up to a value of 65535 will give a length of 2. For larger values the length is 4.

variable name The name of the variable whose length you want. (The variable can be pointer qualified, substringed, or subscripted.)

TYPE	DECLARED CHARACTERISTIC	'LENGTH' RETURNS NUMBER OF:
string constant	character	bytes
	hexadecimal	bits
	bit	bits
arithmetic constant	decimal	2 or 4
	binary	2 or 4
variable	BIT	bits
	not BIT	bytes

Figure 17. Values Returned by LENGTH Built-In Function

Examples of the LENGTH Built-In Function

Some examples of LENGTH used with a string constant are:

```
DCL BUF CHAR(LENGTH('SOME MESSAGE'))
  INIT('SOME MESSAGE');
```

...BUF is CHAR(12).

```
C80(1:LENGTH('MESSAGE')) = 'MESSAGE';
```

...The string 'MESSAGE' is assigned to the first seven bytes of C80.

An example of LENGTH used with an arithmetic constant is:

```
DCL B BIT(LENGTH(42));
```

...The length of B is BIT(2).

Some examples of LENGTH used with a variable name are:

```
DCL ARRAY(10) CHAR(6);
CTR = LENGTH(ARRAY);
```

...CTR is assigned a value of 6.

```
DCL B BIT (4);
FIX = 4 * LENGTH(B);
```

...FIX is assigned a value of 16.

Note on Character String Constants

When one of the characters in the character constant you are coding is an apostrophe, code two consecutive apostrophes. The value returned by LENGTH ignores the extra apostrophe. For example, the value of the character constant 'O''NEIL' is six (bytes).

MAX -- Maximum Value Built-In Function

Purpose

The MAX built-in function returns the arithmetic value that is the largest of the result of the expressions you specify as arguments.

Rules

MAX is not a reserved keyword, but if you explicitly declare MAX as a variable in your program, it loses its built-in function in the program.

Syntax

```
--------MAX-------(-----argument list-----)----------
```

Operands

argument list — Code two or more arithmetic expressions separated by commas.

Examples of the MAX Built-In Function

```
A = MAX (B,C);
```

...The larger of the two arguments, B or C, is assigned to A.

```
IF MAX(A,B*3,C) = Y THEN GO TO STOP;
```

...The largest of the three arguments is compared with Y.

MIN -- Minimum Value Built-In Function

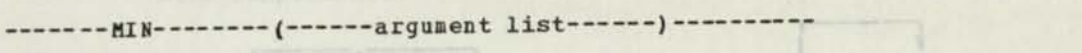
Purpose

The MIN built-in function returns the arithmetic value that is the smallest of the results of the expressions you specify as arguments.

Rules

MIN is not a reserved keyword, but if you explicitly declare MIN as a variable in your program, it loses its built-in function in the program.

Syntax



Operands

argument list	Code two or more arithmetic expressions separated by commas.

Examples of MIN

```
IF MIN (A,B+C,D) = LOWEST THEN GOTO FINISH;
```

...The argument with the smallest value is compared to LOWEST

```
Z = MIN (MAX(A,B),C);
```

...Z is assigned the smaller of the two arguments.

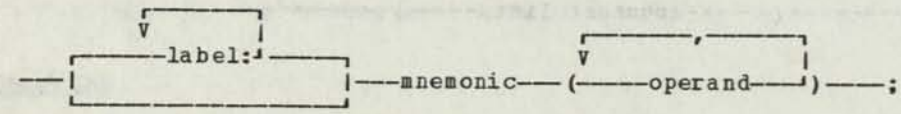
CHAPTER 3. MACHINE INSTRUCTION SUPPORT

The compiler support of machine instructions permits you to code statements whose operations are machine instruction mnemonics or extended mnemonics, and whose operands are PL/DS expressions. This is in contrast with the compiler's DPPX Assembler instruction support (see, MACGEN), which requires operands in formal assembler format.

For details of the machine instructions, refer to the publication, 8100 Information System: Principles of Operation.

For details of the extended mnemonics, refer to the publication, DPPX Assembler Programming: Language Reference and Guide.

SUPPORTED MACHINE INSTRUCTION SYNTAX



Operands

label — Optional. Specify one or more labels, each followed by a colon.

mnemonic — Code the mnemonic for the machine instruction. The mnemonics and extended mnemonics are described on the following pages. This name cannot be used for any other purpose in the procedure being compiled.

operands — The length, number, and type of operands allowed for each instruction are specified in the following pages. Substringing, subscripting, and pointer qualification of operands is allowed.

OPERANDS IN SUPPORTED MACHINE INSTRUCTION STATEMENTS

NUMBER OF OPERANDS

The number of operands required depends on the instruction used. Usually a supported machine instruction will have the same number of arguments as its assembler-language counterpart. However, when the 8100 assumes an implied register number as an argument to the instruction, the supported machine instruction has one more operand than the assembler form. The additional operand is always written first and it is an expression giving the value to be associated with the implied register. The following supported machine instructions have the additional operand:

IOI

JBZ

TYPES OF OPERANDS

The operands for the mnemonics must follow compiler syntax and must be of the length and type required by the instruction, but usually need not be registers or follow register affinity requirements. For exceptions to this, see "Instructions That Require Registers," below.

The operands of a supported machine instruction can be valid PL/DS variables or expressions. Substringing, subscripting, and pointer qualification of operands is allowed. If the hardware instruction requires an operand in a register, and the variable or expression is not in a register, the compiler generates housekeeping code to load/store the value in/from a suitable work register. Apart from loading and storing, the compiler will make no adjustments to the operands specified. Specifically, no padding or truncation of operands is performed. Each operand is in one of the following classes:

value-in-register
immediate
address-in-register
base-displacement

"Value-in-register" Operands

The value of the given expression is to be associated with a suitable register. This association can be done by one of: declaration as a register variable; compiler optimization associating variables with register; compiler code generation issuing housekeeping code before and/or after the instruction. For example:

```
DCL P REGISTER(4)  PTR RSTD;
DCL (X,Y BASED(P))  FIXED(15);
     CTLZ (X,Y);     /* COUNT IN "Y", ANSWER IN "X"*/
```

...might yield the following code:

```
LHN   @02H,P      load "Y" (compiler chose @02H)
CTLZ  @18H,@02H   count leading zero bits in "Y"
STH   @18H,X      save count of leading zeros
STHN  @02H,P      save "Y"
```

If OPTIMIZE is in effect, "X" would probably be assigned to a register, which means that "X" instead of "@18H" would be used in the "CTLZ" and no "STH" is needed following the "CTLZ". The following code would result:

```
LHN   @02H,P      load "Y" (compiler chose @02H)
CTLZ  X,@02H      count leading zero bits in "Y"
STHN  @02H,P      save "Y"
```

"Immediate" Operands

The expression is not to be associated with a register or storage but is to be passed directly to the final assembly phase. For example:

```
DECLARE P PTR,                          /* ASSUME COMPILER PUTS IN REG */
     BV FIXED(8) BASED,                 /* VAL-IN-REG                  */
     KONS BIT(8) CONSTANT('7F'X);       /* IMMEDIATE                   */
LRI (P->BV,                             /* "VAL-IN-REG'                */
     KONS);                             /* "IMMEDIATE"                 */
```

...gives the following:

```
LRI  @01B,X'7F'     target instruction
STN  @01B,P         save "P->BV"
```

"Address-in-Register" Operands

The address of the given operand must be associated with a suitable register prior to issuing the given instruction. As in "value-in-register", above, this implies housekeeping before the instruction. For example:

```
BR (LBL1);     /* "LBL1" IS A LOCAL LABEL */
```

...gives the following

```
LA   @02W,LBL1    "@02W" chosen by compiler
BR   @02W         target instruction
```

Proper declaration will eliminate the need for housekeeping code as in the following example:

```
DECLARE
     XITREG PTR REGISTER(24) RSTD,
     XITLBL LABEL BASED VALUERANGE(*);
BR (XITREG->XITLBL); /* user "XITREG" */
```

...gives the following:

```
BR   XITREG       target instruction
```

"Base-Displacement" Operands

A base displacement form of the operand is required. For example, the second argument of "LHS":

```
DECLARE
     PR  REG(4) PTR RSTD,              /* NOT A BASE REG        */
     XX  FIXED(16) BASED(PR) POS(5),   /* DISP=4                */
     I FIXED(16);                      /* CMPLR ASSIGNS TO REG  */
LHS (I,XX);                            /* USER BII              */
```

Gives the following:

```
LHR  @14H,PR      compiler picks base reg
LHRU @14W, PR
LHS  I,XX(@14W)   target instruction
```

Register Operands

If the instruction requires an operand to be in a register, and the variable you specify is not a register, the compiler will take care of the housekeeping required to load or store the variable. For exceptions to this, see "Instructions That Require Registers," below. There are advantages if you do not explicitly specify registers, but in a few cases this is unavoidable.

The Advantages of Avoiding Register Operands

There are two advantages of avoiding explicit registers except when processing requirements force their use:

1. It prevents the unnecessary restriction of registers, and allows the compiler to allocate registers more efficiently. In extreme cases, if too many registers are restricted, the compiler may run out of registers and be unable to complete the compilation.

2. It allows the optimization phases of the compiler to assign variables to registers according to frequency of use. Thus, even if a variable is

not declared as a register, it may be assigned to one, and the compiler-generated code for the machine instruction will be adjusted accordingly.

Instructions That Require Registers

The following mnemonics are exceptions to the above discussion. They each require a variable declared as a register in the operand listed:

BAL	operand-1
BALR	operand-1
BCTR	operand-1
DHR	operand-1
JAL	operand-1
LHQ	operand-2
MHR	operand-1
STHQ	operand-2

Notes:

1. The MHR instruction has different meanings depending on whether or not the first register number is a multiple of four. See the description in the publication, IBM 8100 Information System: Principles of Operation.

2. The DHR instruction's first register number must be a multiple of four, and both this register and the even-numbered register following it must be restricted and then loaded using conventional assignment statements before issuing the DHR.

Extended Mnemonics

The Control-Immediate and Input/Output instructions have extended mnemonics (for example: KIR and KIW) that indicate whether the instruction is performing a read or a write. A "read" causes a value to materialize in a register, which implies that store code follows the supported machine instruction. A "write" implies that a value exists in a register to be written and therefore there must be code to load a register prior to issuing the supported machine instruction.

Note: The compiler's optimization phase frequently assigns the given expression to a register so that load/store code is not needed.

COMPILER ADJUSTMENTS TO VARIABLES IN ARGUMENTS

If the supported machine instruction requires an address (such as BCR), the compiler generates instructions to load the address of the variable you specify into a register.

Apart from loading and storing, the compiler makes no adjustments to the variables specified. In particular, the compiler performs no padding or truncation of variables.

If the instruction requires an indirect operand (such as LHNI), code a pointer to the item. The compiler causes this pointer to be adjusted if the instruction increments or decrements.

SUPPORTED MACHINE INSTRUCTIONS AND EXTENDED MNEMONICS

The following table lists the supported machine instructions, operands, and names.

Note that the compiler supports some extended mnemonics (for example, BER and KIR) in addition to the actual machine instructions.

The following explains the operand symbols used in the table below.

Each operand will be a valid PL/DS expression, but will have constraints on its attributes.

Operand	Meaning
B	operand must be FIXED(8) or CHAR(1) or BIT(8) on a byte boundary
E	operand must be ENTRY
H	operand must be FIXED(15) or FIXED(16) or CHAR(2) or BIT(16) on a halfword boundary
In	operand must be immediate value (which can be represented in n bits)
L	operand must be LABEL
P	operand must be POINTER or FIXED(32)
RS	operand must be FIXED(16) or BIT(16) (user has responsibility to ensure that the operand contains a valid register space address)
Z	operand must be the constant 0
*	operand attributes are not checked
xA	operand must be accessible via address-in-register
xD	operand must be accessible via base-displacement
xL	operand must have attribute LOCAL
xO	operand is an output which implies that neither constants nor arithmetic expressions nor string expressions are permitted.
xR	operand must be declared as register
xU	operand must be unsigned
xZ	zero used if explicitly specified (not a value of zero to be associated with some register)
cc	opcode is one of extended branch mnemonics:

Z	E	O	TE
P	H	X	null
M	L	NO	NM
NZ	NE	NX	
NP	NH	NL	
Y	V		

Note: The control immediate (KI) machine instruction must be used with care. See the publication, 8100 Information System: Principles of Operation.

The PL/DS compiler does not recognize or interpret the specialized meanings of control immediate operands, and does not act on the values preceding and following the control immediate statement in a "knowing" way.

MNEMONIC	ext = extended	OPERANDS			NAME
AHR		HO	H	–	ADD (halfword, register)
AHRI		HO	I4	–	ADD (halfword, register-immediate)
AR		BO	B	–	ADD (byte, register)
ARI		BO	I8	–	ADD (byte, register-immediate)
AYHR		HO	HZ	–	ADD WITH CARRY (halfword, register)
AYHRE		PO	PZ	–	ADD WITH CARRY (halfword, register, extended)
AYR		BO	B	–	ADD WITH CARRY (byte, register)
BAL		PRO	E	–	BRANCH AND LINK
BALR		PRO	EA	–	BRANCH AND LINK (register)
BALRZ	ext	PRO	Z	–	BRANCH AND LINK (register with second arg 0; load current address without branching)
BccR	ext	LA	–	–	BRANCH ON CONDITION (register)
BC		I4	L	–	BRANCH ON CONDITION
BCR		I4	LA	–	BRANCH ON CONDITION (register)
BCTR		BRO	LA	–	BRANCH ON COUNT (byte, register)
BNX		B	L[1]	–	BRANCH ON INDEX (byte)
CHR		H	H	–	COMPARE (halfword, register)
CLHS		PO	PO	HO	COMPARE LOGICAL (halfwords, storage)
CLS		PO	PO	HO	COMPARE LOGICAL (bytes, storage)
CR		B	B	–	COMPARE (byte, register)
CTLZ		HO	HO	–	COUNT LEADING ZEROS (halfword)
CYHRE		P	PZ	–	COMPARE WITH CARRY (halfword, register, extended)
DHR		HRUO	HU	–	DIVIDE (halfword, register)
IO		BO	H	–	INPUT/OUTPUT (byte)
IOH		HO	H	–	INPUT/OUTPUT (halfword)
IOHR	ext	HO	H	–	INPUT/OUTPUT (halfword, read)
IOHW	ext	H	H	–	INPUT/OUTPUT (halfword, write)
IOI		B[2]	BO	I8	INPUT/OUTPUT (byte, immediate)
IOIR	ext	B	BO	I8	INPUT/OUTPUT (byte, immediate, read)
IOIW	ext	B	B	I8	INPUT/OUTPUT (byte, immediate, write)
IOR	ext	BO	H	–	INPUT/OUTPUT (read)
IOW	ext	B	H	–	INPUT/OUTPUT (write)
JAL	ext	PRO	LL	–	JUMP AND LINK
JBZ		H[2]	I4	LL	JUMP ON BIT ZERO (halfword)
Jcc	ext	LL	–	–	JUMP ON CONDITION
JC		I4	LL	–	JUMP ON CONDITION
JCX	ext	I4	LL	–	JUMP ON CONDITION EXTENDED
KI		BO	I8	–	CONTROL IMMEDIATE
KIR	ext	BO	I8	–	CONTROL IMMEDIATE (read)
KIW	ext	B	I8	–	CONTROL IMMEDIATE (write)
KIZ	ext	Z	I8	–	CONTROL IMMEDIATE (zero as register)
KDO		I4	–	–	CONTROL DIRECT OUT
L		BO	BD	–	LOAD (byte)
LA		PO	*D	–	LOAD ADDRESS
LAT		PO	RSO	–	LOAD FROM ADDRESS TRANSLATION TABLE
LH		BO	BD	–	LOAD (halfword)
LHN		HO	P	–	LOAD (halfword, with index)
LHND		HO	PO	–	LOAD (halfword, with index decremented)
LHNI		HO	PO	–	LOAD (halfword, with index incremented
LHQ		I2	PRO	–	LOAD (halfwords, quadrant)

Figure 18. Table of Supported Machine Instructions (Part 1 of 2)

[1] See BNX note below.
[2] Implied register value (described above, under "Number of Operands.")

MNEMONIC ext = extended	OPERANDS			NAME
LHR	HO	H	–	LOAD (halfword, register)
LHRLU	HO	P	–	LOAD (halfword, register, lower half from upper)
LHRN	HO	RS	–	LOAD (halfword, register-indirect)
LHRU	PO	P	–	LOAD (halfword, register, upper half)
LHRUL	PO	H	–	LOAD (halfword, register, upper half from lower)
LHS	HO	HD	–	LOAD (halfword, short form)
LN	BO	P	–	LOAD (byte, with index)
LND	BO	PO	–	LOAD (byte, with index decremented)
LNI	BO	PO	–	LOAD (byte, with index incremented)
LR	BO	B	–	LOAD (byte, register)
LRI	BO	I8	–	LOAD (byte, register-immediate)
LRN	BO	RS	–	LOAD (byte, register-indirect)
LW	PO	PD	–	LOAD (word)
MHR	HRUO	HU	–	MULTIPLY (halfword, register)
MVS	PO	PO	HO	MOVE (bytes, storage)
MVHS	PO	PO	HO	MOVE (halfwords, storage)
NHR	HO	H	–	AND (halfword, register)
NR	BO	BZ	–	AND (byte, register)
NRI	BO	I8	–	AND (byte, register-immediate)
OHR	HO	H	–	OR (halfword, register)
OR	BO	BZ	–	OR (byte, register)
ORI	BO	I8	–	OR (byte register-immediate)
PC	–	–	–	PROGRAM CHECK ('FFFF'X)
RL	BO	I3	–	ROTATE LEFT (byte)
RLH	HO	I4	–	ROTATE LEFT (halfword)
SHR	HO	H	–	SUBTRACT (halfword, register)
SHRI	HO	I4	–	SUBTRACT (halfword, register-immediate)
SLHL	HO	I4	–	SHIFT LEFT (halfword, logical)
SLL	BO	I4	–	SHIFT LEFT (byte, logical)
SR	BO	B	–	SUBTRACT (byte, register)
SRI ext	BO	I8	–	SUBTRACT (byte, register-immediate)
ST	B	BDO	–	STORE (byte)
STAT	P	RSO	–	STORE TO ADDRESS TRANSLATION TABLE
STH	H	HDO	–	STORE (halfword)
STHN	H	P	–	STORE (halfword, with index)
STHND	H	PO	–	STORE (halfword, with index decremented)
STHNI	H	PO	–	STORE (halfword, with index incremented)
STHQ	I2	PRO	–	STORE (halfwords, quadrant)
STHRN	H	RS	–	STORE (halfword, register-indirect)
STHS	H	HDO	–	STORE (halfword, short form)
STN	B	P	–	STORE (byte, with index)
STND	B	PO	–	STORE (byte, with index decremented)
STNI	B	PO	–	STORE (byte, with index incremented)
STRN	B	RS	–	STORE (byte, register-indirect)
STW	P	PDO	–	STORE (word)
SYHR	HO	HZ	–	SUBTRACT WITH CARRY (halfword, register)
SYHRE	PO	PZ	–	SUBTRACT WITH CARRY (halfword, register, extended)
SYR	BO	B	–	SUBTRACT WITH CARRY (byte, register)
TRI	B	I8	–	TEST (byte, register-immediate)
TS	BZ	BO	–	TEST AND SET (byte)
XHR	HO	H	–	EXCLUSIVE OR (halfword, register)
XR	BO	B	–	EXCLUSIVE OR (byte, register)
XRI	BO	I8	–	EXCLUSIVE OR (byte, register-immediate)

Figure 19. Table of Supported Machine Instructions (Part 2 of 2)

EXAMPLES OF SUPPORTED MACHINE INSTRUCTION USE

EXAMPLE OF ARITHMETIC INSTRUCTIONS

This example is typical of the arithmetic built-in instructions.

```
DCL (X,Y) FIXED(15);
AHR(X,Y);
```

This has very nearly the same effect as: X = X + Y; and will work whether or not X and Y are in STATIC, AUTOMATIC, BASED or PARAMETER, or have been assigned to registers by the compiler's optimization phases. The only differences between the built-in instruction and the assignment statement are that:

- the built-in instruction guarantees that an AHR will be generated.
- the condition code set as a result of the AHR will be preserved even if code follows the AHR to store the result in X.

EXAMPLE OF BRANCH AND JUMP INSTRUCTIONS

This example is typical of the branch and jump supported machine instructions. Note that the operands are more complex than in the preceding example.

```
DCL X FIXED(15) BASED(P);
DCL Y(10) FIXED(15);
DCL J FIXED(8);
DCL (P,Q) POINTER;
AHR(Q->X,X+Y(J));
BZR(LAB);
X=-X;
LAB:
```

The built-in instruction BZR causes a branch to be taken to the label LAB if the result of the AHR is zero. Even though LAB is accessible by means of a jump instruction, a BZR instruction will be generated. This could be useful to avoid generation, by the compiler, of the assembler's G instruction, which may expand into a long jump.

If a branch or jump built-in instruction is issued, the user should ensure that the condition code is set correctly by setting it with a built-in instruction. For example, the following code is not safe, because the condition code is not guaranteed. (If the expression X + Y occurs earlier, it may be retained in a register to avoid recalculation, and addressing X in order to store the result may require addition of an offset to a base value.)

```
DCL (X,Y) FIXED(15);  /***** unsafe coding *****/
X = X + Y;            /***** to illustrate *****/
BZR(LAB);             /***** above text    *****/
LAB:
```

EXAMPLE OF REGISTER USE IN BUILT-IN INSTRUCTIONS

Operands of built-in instructions do not generally have to be declared registers. The compiler generates the housekeeping instructions to load, store, and evaluate arguments. The specified instruction is generated. The following example shows the code generated when the operands of AHR are: a) registers, b) complex expressions.

```
* DCL 1 S BASED(P),
*       2 A FIXED(15),
*       2 B(10) FIXED(15);
* DCL P PTR;
* AHR(X,Y);
          AHR   X,Y
* AHR(P->B(6),P->A);
          LHS   @0V00007,B+10(P)
          LHN   @02H,P
          AHR   @0V00007,@02H
          STHS  @0V00007,B+10(P)
```

EXAMPLE OF CODING REGISTERS IN BALR AND BCR:

BALR requires that the first operand be a pointer register and the second be of type ENTRY. This example shows how it should be coded:

```
DCL ENT1     ENTRY EXTERNAL;
DCL RETPTR   PTR REGISTER(4);
DCL P        PTR;
DCL BENTRY   BASED ENTRY VALRG(*);
P=ADDR(ENT1);              /* P HAS TARGET FOR BALR */
 - - -
BALR(RETPTR,P->BENTRY);
```

BCR requires that the second operand be of type LABEL:

```
DCL BLAB     LABEL BASED VALRG(LAB1,LAB2);
DCL P        PTR;
P=ADDR(LAB1);
 - - -
BCR(8,P->BLAB);
```

EXAMPLE OF CODING BNX

Because of the need to provide flow information on BNX instructions, the VALUERANGE information must be associated with the BNX instruction. This enables the compiler to know the possible target points for the BNX.

The following example shows how a BNX instruction should be coded:

```
DECLARE
  BNXTAB(4) FIXED(16)           /* BRANCH TABLE FOR BNX */
       STATIC LOCAL
       INIT( ADDR(L0),          /* LABEL FOR INDEX=0    */
             ADDR(L1),          /* LABEL FOR INDEX=1    */
             ADDR(L2),          /* LABEL FOR INDEX=2    */
             ADDR(L3)),         /* LABEL FOR INDEX=3    */
  BNXLAB LABEL                  /* LABEL FOR BNX BII    */
       BASED(ADDR(BNXTAB(1)))   /* USE BNXTAB           */
       VALUERANGE(L0,L1,L2,L3),
  XFORBNX FIXED(8);             /* INDEX FOR BNX        */

     BNX                        /* A BNX BII            */
       (XFORBNX,                /* THE INDEX            */
        BNXLAB );               /* USES BNXTAB          */
```

CHAPTER 4. PL/DS MACRO OUTER CODE

The PL/DS macro code enables you to code preprocessor-like logic that is executed by the single-pass macro processing phase at the beginning of the PL/DS compiler run. After this macro phase the macro statements disappear from the source text that is passed to the compile phase of the compiler.

There are two macro environments: the macro outer environment and each macro definition's environment. This chapter describes macro outer code and macro invocations.

The PL/DS macro language is versatile, with instructions for branching and conditional execution, for example. With the macro code you make changes to the compile code before forwarding it to the compile phase. Your macro logic is an efficient way to "customize" standardized code with the coding of parameters that reflect the changes you want incorporated in this version of the program.

The macro code you use is referred to here as macro "outer" statements. This distinguishes it from macro definition code. Macro definitions are PROC-like groups of macro definition statements. You cause macro definitions to execute during the macro processing phase with macro invocations, also described in this chapter.

Besides branching and conditional macro statement execution, you can alter character strings in the compile source code. We call these compile code replaceable strings "target strings."

Target strings are strings in your non-macro code that match the names of variables you declare in your macro code. When the conditions are right -- a target string is "eligible" and the macro outer variable with the same name is "activated" -- the macro phase replaces the target string with the characters of whatever value the macro outer variable is presently assigned. The macro processing phase performs one sequential scan of your input data set. The macro statements can be interspersed with the compile statements; the compile phase scans all statements and executes each macro statement as it encounters it. Therefore, you must place in your source data set the macro variable DECLARE, ACTIVATE, and assign statements before the target strings you want them to affect.

You can disable the target string replacement by any macro variable with the DEACTIVATE statement. You can re-enable a deactivated variable with the ACTIVATE statement.

Besides changing the contents of character strings with macro outer code, you can also delete statements from the source code data set. See the %GOTO statement description. If for any reason you want to bypass the macro processing phase in your compiler run, see the compiler option, "NOMACRO."

You can use the %INCLUDE statement to bring additional source code in from a user library for processing during the macro phase.

There are some references in the macro outer code descriptions to "answer text." Answer text is lines of source code inserted into the source data set under certain conditions by certain macro definitions you invoke. It is described in Chapter 5 of this publication.

You need to understand the meaning and uses of three parts of macro language:

- macro definition
- macro execution
- macro outer statements

Macro Definition: You define or create a macro by specifying its name and a set of macro definition statements that are known by this name. Macro definitions may exist within your program, or you can catalog them for use in more than one program by putting them in a library data set.

The rules for coding macro definitions are described separately in Chapter 5 of this publication.

Macro Execution: When you invoke a macro definition in your program, the statements specified in that macro's definition are executed. The execution of macro definitions alters the values of shared (EXTERNAL) variables, and produces lines of source code, called answer text, which replace the macro invocation.

Macro Outer Statements: You may code statements that control the sequence and flow of macro processing and that perform source statement modifications outside of macros. The compiler macro statements that you use for this are called Macro Outer Statements.

SHARED VARIABLES

You can share the assigned value of a macro outer variable between the macro outer environment and the invoked macro definition environment by declaring it EXTERNAL in both places. The ACTIVATED or DEACTIVATED status is not shared, and must be separately specified in each environment.

In either of these environments, if the same variable name is declared as INTERNAL, the macro processing phase treats them as two different variables.

MODIFYING SOURCE TEXT WITH MACRO OUTER CODE

Macro outer code modifies source text before it is compiled. Modifications to source text consist of:

- Replacing target strings in non-macro text with their macro variable assigned values.
- Deleting segments of source text.
- Adding segments of source text.

REPLACING TARGET STRINGS

Replacement of character strings in non-macro text occurs when a target string is present in a non-macro statement. The target string is replaced by the value earlier assigned to its corresponding macro variable with a macro assignment statement.

> Note: The macro processing phase of the compiler is a single pass through the source text dataset. This pass proceeds sequentially from beginning to end of the source code except when macro statements like %GOTO cause input to be skipped over.
>
> You must code your macro statements so that in this single pass the compiler will have scanned the macro code that assigns your macro variable values before it scans the non-macro code containing the target strings that you want replaced.

An eligible target string in non-macro text will be replaced unless its corresponding macro variable has been specified in a DEACTIVATE macro statement. See the description of the ACTIVATE and DEACTIVATE macro statements.

Any target string in the arguments and parameters you code in a macro invocation are replaced by their activated macro outer variable's present assigned value before the invoked macro is executed.

Eligible Target Strings: Certain character strings are not eligible target strings:

- Strings in comments or within string constants.
- Strings that correspond to a macro variable but are immediately preceded or followed by a letter, digit, or apostrophe.
- Character sequences that correspond to a macro variable except for imbedded blanks not present in the macro variable.
- Macro variables specified in the DEACTIVATE statement.

For example, the macro target string B is not replaced in the following cases:

- In the assignment: X = 'ABCD'; (it is within a string constant).
- In the strings: AB, 101B, or '1'B. (Preceded by letter, digit, apostrophe.)

A character target string might be replaced by a value that contains another target string. Because of this possibility, macro phase processing rescans the replaced string to see if it contains an activated variable. If it does, macro phase processing replaces this string with its corresponding variable's assigned value. Then the macro phase rescans again. If the replacement also contains a target string, it is further replaced. This re-rescanning continues until there is no replacement for the string.

If the value that replaces the target string contains one or more embedded blanks, the re-rescanning is confined to the individual parts of the replacement string separated by the blanks, starting from the left. Given the following variables and assigned values:

```
%DCL A...;
%A = 'JO';
%DCL B...;
%B = 'O E';
%DCL JO...;
%JO = 'B E';
%DCL E...;
%E = 'Z';
```

...the following target string:

```
A
```

...after scanning and rescanning yields:

```
O Z Z
```

The assigned value of a macro variable should not contain the name of the macro variable it is replacing. If it does, a replacement loop is caused because macro phase processing replaces the target variable with the same macro variable.

DELETING SOURCE TEXT

Deletion of source text occurs when a macro GOTO statement directs macro processing to stop scanning and resume processing at a particular labeled statement further on. Text between these two points does not appear as part of the modified text that will be compiled.

ADDING TEXT TO THE SOURCE TEXT

Addition of source text occurs when a macro %INCLUDE statement is encountered. The requested source text is added from a user library at this point, is scanned by the macro processing phase, and the result becomes part of the modified source that is compiled.

IDENTIFIER LENGTH IN MACRO CODE

In PL/DS macro code, macro variable names of up to 16 characters are permitted.

COMPILER PROCESSING OF MACRO CODE

All macro definitions, macro invocations, GOTOs, and the rest of the macro code are processed and executed during the one-pass macro processing phase. The macro processing phase leaves no macro code for the compile phase, which comes next. If your macro logic results in any change to the compiler source code that looks like macro language, it will cause an error in the compile phase.

The inputs and results of the macro phase are shown in Figure 20.

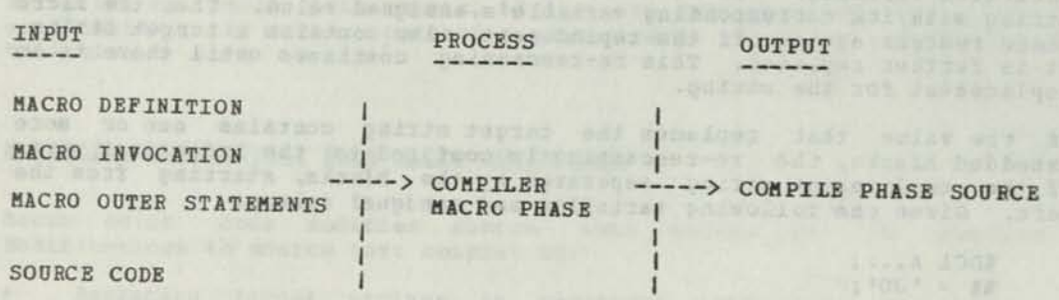


Figure 20. Macro Processing Phase Operation

The macro phase scans the source text and forms a modified source text. Macro statements are used to tell the macro phase how to modify the source text. These macro statements will not themselves appear in the modified text. If a macro invocation is in the source, the statements in the definition of that macro are brought in from a library and processed with other macro statements. The macro invocation statement does not appear in the modified source text.

The modified source serves as input to the compile phase.

The macro facility makes available the following modifications to your source text before it is compiled into assembler language code:

- Change segments of the source text.
- Delete segments of the source text.
- Add segments of text to your source text.

COMPILER SOURCE MODIFICATION SEQUENCE

During the one-pass macro processing phase, the compiler immediately rescans each just-changed target variable. On the rescan it is looking for the presence of any additional target variable.

The compiler replaces any target variable it finds on the rescan, and then rescans the results of the replacement only for any additional eligible target variable. (See the note in the description of the concatenation operator, below.)

Only when the rescan does not yield another target variable does the compiler macro phase proceed to scan the following text.

COMPILER IDENTIFICATION OF MACRO LANGUAGE CONTENTS

Identification of Macro Definitions and Macro Definition Statements

The compiler recognizes your macro definitions by the %MACRO and %END statements that begin and end them. The macro definition statements themselves have no special character prefixes, but their presence between %MACRO and %END identifies them.

Identification of Macro Invocations

The compiler recognizes each macro invocation by the question mark you code in the first column (within margins) of the statement.

Identification of Macro Outer Statements

The compiler recognizes your macro outer statements by the percent sign in the first column (within margins) of the statement. The %MACRO and %END statements are special cases of macro outer statements, described in the macro definition chapter.

Identification of Non-Macro Text

Non-macro text is input that is neither macro statements nor macro definitions. No line of input can intermix macro statements and non-macro text. The presence of a "%" as the first (non-blank) character on a line indicates that this line can consist solely of macro statements. For example, the following would be an error:

```
%DECLARE A FIXED; X=0;
```

Similarly, it is an error to have a macro statement in a line of non-macro text. For example, the following would be an error:

```
A=B+C; %DECLARE X CHAR;
```

Note: Do not confuse the single "%" with the "%%" concatenation operator, described below.

MACRO OUTER ENVIRONMENT KEYWORDS

Figure 21 lists all of the words recognized by the compiler macro processing phase as macro outer environment keywords. Those keywords that are indicated as reserved cannot be used for variable names.

KEYWORD	ABBREVIATION	RESERVED
ACTIVATE	ACT	Yes
CHARACTER	CHAR	
DEACTIVATE	DEACT	Yes
DECLARE	DCL	Yes
ELSE		Yes
END		Yes
EXTERNAL	EXT	
FIXED		
GOTO	GO TO	Yes
IF		Yes
INCLUDE		Yes
INTERNAL	INT	
MACRO		Yes
THEN		Yes

Figure 21. Macro Outer Keywords

COMPILER CONTROLS FOR MACRO PROCESSING

MACRO SOURCE CODE MARGINS

The macro outer statements in your source data set must observe the left and right source columns specified in the MARGINS compiler option. If there is any non-macro text outside of these columns, the compiler copies it to the output text of the macro phase. See "Compiler Options."

SUPPRESSING MACRO PROCESSING

You can prevent compiler macro processing phase with the NOMACRO compiler option. See "Compiler Options."

THE CONCATENATION OPERATOR: %%

The double "percent", %%, is an operator you can use in non-macro text to concatenate variables, especially target variables. During the macro processing phase, the compiler replaces %% with a null string, bringing together the strings on either side.

For example, during the macro phase the last statement in this set:

```
% DCL (XX, YY, ZZ) CHAR;
% XX = 'AB';
% YY = '01';
% ZZ = '02';
    XX%%YY = XX%%ZZ + 3;
```

...becomes:

```
AB01 = AB02 + 3;
```

The following example illustrates how the name prefix of a structure can be parameterized using %%. Given this source code:

```
File DCB COPY:
     %ACT Z;
     %IF Z='' %THEN
        %Z='DCB';
     DCL 1 Z%%1 BDY(HWORD) BASED,
           3 Z%%F1   FIXED(15),
           3 Z%%F2   CHAR(2),
             - - -
           3 Z%%FN;
     %Z='';
     %DEACT Z;
```

...the statement

```
%INCLUDE SYSLIB(DCB);
```

...would return:

```
DCL 1 DCB1 BDY(HWORD) BASED,
      3 DCBF1   FIXED(15),
      3 DCBF2   CHAR(2),
        - - -
      3 DCBFN;
```

...but the statement

```
%Z='ABC';
%INCLUDE SYSLIB(DCB);
```

...would return:

```
DCL 1 ABC1 BDY(HWORD) BASED,
      3 ABCF1   FIXED(15),
      3 ABCF2   CHAR(2),
        - - -
      3 ABCFN;
```

Note: As explained elsewhere, target strings that have been replaced are rescanned for possible additional target strings. The string that results from %% concatenation is not rescanned in its entirety. Only each individual token is rescanned. The concatenation can be thought of as taking place only when all rescanning of the tokens on each side of it is completed.

In the following example, these macro variable assignments are in effect:

```
% A  = 'X';
% B  = 'Y';
% XY = 'ZZ';
% X  = 'M';
% Y  = 'N';
% MN = 'EE';
```

...the target string scan and rescan of this non-macro text:

```
A%%B
```

...would result in:

```
MN
```

SUMMARY OF MACRO OUTER STATEMENTS

The following lists the macro outer statements described in this section.

STATEMENT	USE
%ACTIVATE	Causes the specified macro variable's assigned value to replace eligible matching target strings.
%assignment	Assigns a value to a macro variable.
%DEACTIVATE	Blocks the specified macro variable's assigned value from replacing eligible matching target strings.
%DECLARE	Defines the data type and scope of a macro variable.
%GOTO	Changes the point at which scanning continues.
%IF	Provides conditional macro statement execution.
%INCLUDE	Includes source text from a user library.
%null	Causes no action.

%ACTIVATE -- Activate Macro Outer Variable

Purpose

You use the macro outer %ACTIVATE statement to restore a previously deactivated macro outer variable to "activated" status.

(Note that there is also a macro definition ACTIVATE statement, for macro definition variables.)

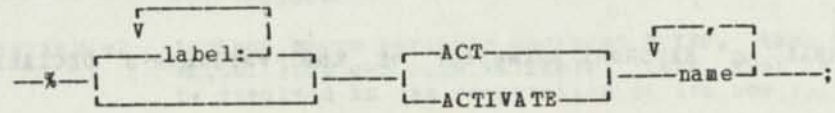
Rules

The default state of each macro outer variable declared INTERNAL is activated, until it is explicitly deactivated.

The default state of each macro outer variable declared EXTERNAL is activated, unless the most recently invoked macro definition that declared the same variable name EXTERNAL also deactivated it.

During the macro processing phase of the compiler, ACT and ACTIVATE are reserved keywords; do not use them as macro variables.

Syntax



Operands

label — Optional. Code one or more labels, each followed by a colon.

name — The name(s) of the macro variable(s) whose eligible corresponding target string(s) will be replaced by their variable(s) assigned value(s).

%Assignment -- Specify Macro Outer Variable Value

Purpose

You use a %assignment statement to specify a value for a macro outer variable.

Rules

The value or values you assign must be the same data type as you declared for the macro variable.

Rules of Target String Replacement by Macro Outer Variable Values

The characters of the value that you assign to the macro outer variable in the macro assignment statement are the characters that the macro phase uses to replace any eligible non-macro target string that matches the name of the macro outer variable. The eligibility of a target string for replacement by the value you assigned its matching macro outer variable is controlled by several factors:

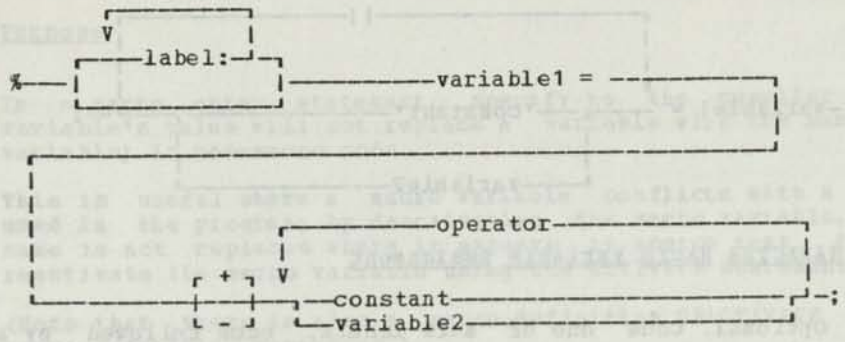
1. The ACTIVATE and DEACTIVATE statements.
2. The RESCAN and NORESCAN parameters of the macro definition ANSWER statement.
3. The INTERNAL or EXTERNAL parameter of the variable's DECLARE statement.
4. The macro processing phase makes a single pass through the source code data set. It must process your macro variable assignment statement before it scans the corresponding target string. (The sequence of scanning can deviate from the actual source statement sequence under control of macro GOTO statements.)
5. The target string is in non-macro text, answer text, or the arguments in a macro invocation. It is not in comments (/* */) or in string constants, and does not have a letter, digit, or apostrophe either just before or just after it.

The value that you assign to a macro variable should not contain unmatched comment delimiters (/* */) or unmatched string delimiters (' '), unless the target variable that it replaces is concatenated (%%) to another target variable that will have the matching delimiters. An unmatched delimiter causes the compile phase to continue scanning in the search for a closing delimiter.

The only macro code that a character constant in an assigned value can contain is a macro invocation.

Apostrophes enclosing character constants do not appear in the value assigned to the receiver, nor are they counted in the length.

Syntax for Assigning a FIXED Macro Variable



Operands for FIXED Macro Variable Assignment

label — Optional. Specify one or more or more labels, each followed by a colon.

variable1 — Specify the FIXED macro variable that is to have assigned to it the value you are coding after the equal sign.

= — The equal sign identifies this as an assignment statement.

constant — Must be a decimal number in the range -2147483647 to +2147483647.

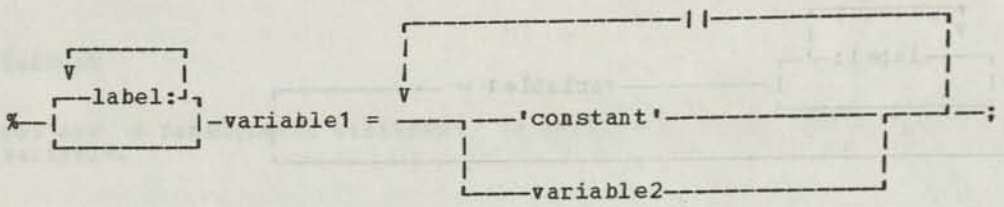
variable2 — Another macro variable declared FIXED, that has an assigned value. (You can code variable1 here, when its present value is involved in the computation of its new value.)

operator — Specify the operation to be performed to compute the value of the receiver.

Use one of: + - * / // (<-remainder)

Evaluation of a source expression that contains operators is done by priority of the operators. First, the prefix operators, then *, /, // operators, and then + and - operators. If you want to change the order of evaluation, you may use parentheses. Operators and their associated operands enclosed in parentheses are evaluated first and are evaluated by priority.

Syntax for Assigning a CHARACTER Macro Variable



Operands for CHARACTER Macro Variable Assignment

label — Optional. Code one or more labels, each followed by a colon.

variable1 — Specify the CHARACTER macro variable (receiver) that is to have assigned to it the value (source) you are coding after the equal sign.

= — The equal sign identifies this as an assignment statement.

'constant' — Code a character string, enclosed in apostrophes.

variable2 — Another macro variable declared CHAR, that has an assigned value. (You can code variable1 here, when its present value is involved in the computation of its new value.)

|| — You can assign a string that contains a concatenated series of functions and/or character constants and/or macro variables.

Notes:

1. Character constants, enclosed in apostrophes, and CHARACTER macro variables can appear in the source expression of the macro assignment statement. The receiver is assigned both the value and the length of the source expression. Apostrophes enclosing character constants do not appear in the value assigned to the receiver. The value assigned to the receiver should not contain unmatched comment or string delimiters.

2. If a character constant is used, it cannot contain macro statements. (The rescan will ignore them and the compile phase will treat them as errors.) The character constant can be a null string (e.g. %B= '';). In this case the receiver is assigned a length of zero.

3. You can concatenate character constants and macro variables with other character constants and macro variables by using the concatenation operator (||). This operator simply connects the operands together; a string is formed with a length equal to the sum of the lengths of the operands. The total length cannot exceed 1,000 bytes.

%DEACTIVATE -- Deactivate Macro Outer Variable

Purpose

In a macro outer statement, specify to the compiler that a macro variable's value will not replace a variable with the same name (target variable) in non-macro code.

This is useful where a macro variable conflicts with a variable name used in the program; by deactivating the macro variable, the variable name is not replaced where it appears in source text. Later, you can reactivate the macro variable using the ACTIVATE statement.

(Note that there is also a macro definition DEACTIVATE statement, for macro definition variables.)

Rules

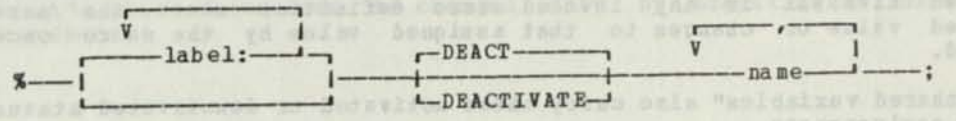
The default state of each outer macro variable declared INTERNAL is activated, until it is explicitly deactivated.

The default state of each macro outer variable declared EXTERNAL is activated, unless the most recently invoked macro definition that declared the same variable name EXTERNAL also deactivated it.

If a DEACTIVATE statement is not scanned (because of the action of a macro GOTO statement), the variable remains activated.

During the macro phase, DEACT and DEACTIVATE are reserved keywords; do not use them as macro variables.

Syntax



Operands

label — Optional. Code one or more labels, each followed by a colon.

name — The name(s) of the macro variable(s) that you are deactivating.

%DECLARE -- Declare a Macro Outer Variable

Purpose

In macro language, assign attributes of FIXED or CHARACTER and INTERNAL or EXTERNAL to a macro variable. (Note that there is also a macro definition DECLARE statement, whose INTERNAL macro variables are known only within the macro definition they are coded in.)

Rules

You cannot assign a precision for FIXED macro variables or a length for CHARACTER macro variables. A FIXED macro variable is represented internally as an assigned binary word with an initial value of zero. A CHARACTER variable has an initial length of zero, which changes as values are assigned to it.

You cannot initialize the macro variable in this statement. Use the %assignment statement.

The DECLARE statement for a macro variable must come before any use of that macro variable. Any macro variable that is not declared is assigned a data type of CHARACTER; the default scope attribute is INTERNAL.

You can declare more than one macro variable on the same macro DECLARE statement. If a number of macro variables have the same attributes, separate the variables with commas and enclose the list in parentheses; follow the list with the attributes. Be sure that a branch at a %GOTO statement does not cause a macro DECLARE statement to be skipped.

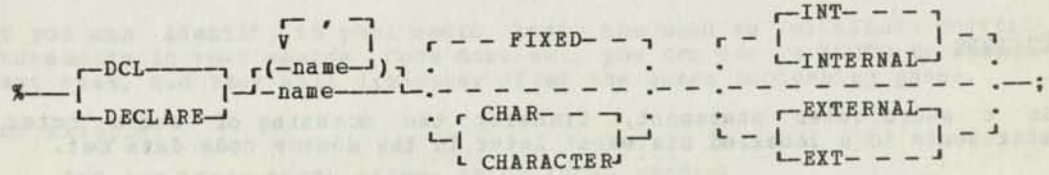
Macro variables declared INTERNAL in a macro outer %DECLARE statement are not known to any macro definition invoked in the same compiler run.

A macro outer variable declared EXTERNAL and a variable of the same name declared EXTERNAL in any invoked macro definition share the same assigned value or changes to that assigned value by the macro once invoked.

Such "shared variables" also carry their activated or deactivated status across environments.

DCL and DECLARE are reserved keywords; do not use them as macro variables.

Syntax



Operands

name — Code the name of the macro variable you are declaring.

The maximum length of the name is 16 characters. The first character must be alphabetic (A-Z). The remaining characters must be alphabetic (A-Z), numeric (0-9), or underscore (_).

FIXED — The macro variable has the attribute, fixed.

CHARACTER/CHAR — The macro variable has the attribute, character. The default is CHARACTER.

INTERNAL/INT — The macro variable has the scope attribute, internal. This is the default scope attribute. INTERNAL macro variables are not known to any macro definition that is invoked during the macro phase.

EXTERNAL/EXT — The macro variable has the scope attribute, external.

EXTERNAL macro variables can be referenced outside of and within a macro definition.

The first time the variable is declared, the value of a FIXED variable is zero and the value of a CHARACTER variable is a null string. Thereafter, the value of the variable is the most recent value assigned to it; this value can be changed only by executing an assignment statement.

%GOTO -- Transfer Control In Macro Outer Code

Purpose

In a macro outer statement, transfer the scanning of macro outer statements to a labelled statement later in the source code data set.

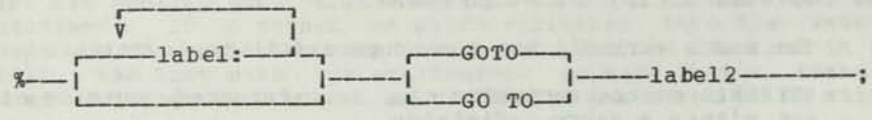
Rules

The statement that scanning is transferred to must be somewhere after the %GOTO statement in your input stream.

Source data set statements between the macro %GOTO statement and the transfer point are ignored; they are not scanned or executed during the macro phase, nor are they compiled during the compile phase. This applies to all macro statements including the DEACTIVATE and ACTIVATE statements.

GOTO and GO and TO are reserved keywords; do not use them as variable names.

Syntax



Operands

label — Optional. Code one or more labels, each followed by a colon.

label2 — The label of the macro statement where scanning is to continue. Be sure that a %GOTO does not unintentionally cause a branch that skips a macro DECLARE.

Note on Transferring Scanning to a Macro Invocation: The label following the trigger character (?) of a macro invocation cannot be the target of a macro %GOTO statement. If you want to transfer the scan to an invocation, you must use a null statement ahead of the invocation. For example:

```
% GOTO MLABEL;
    - - -
    - - -
%MLABEL: ;

          ? LABELS: MACRO1;
```

The label2 parameter of the macro %GOTO statement should be the null statement's label, and not one of the labels following the trigger character. The invocation, ?LABELS:MACRO1; must be on the line after the null statement.

Deleting Source Statements With %GOTO

If you can identify in your macro logic the need to eliminate certain statements in your source code data set, you can use a %GOTO to branch past them, and they will disappear after the macro processing phase.

For example:

```
%IF A=B %THEN %GOTO FIRST; %ELSE %GOTO SECOND;
%FIRST ...┐
 - - -    |
 - - -     > statements not deleted if A=B
 - - -    |
 - - -    ┘
%SECOND ...
 - - -
 - - -
 - - -
```

%IF, %THEN, %ELSE Conditional Macro Outer Statement Execution

Purpose

States a condition during macro processing under which an accompanying %THEN or optional %ELSE clause is to be executed.

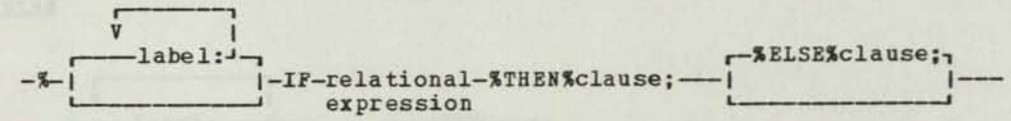
Rules

If there is no %ELSE clause, and the %IF condition is false, processing continues with the source text following the %THEN clause.

Multiple IF relational expressions connected by | or & are not supported in PL/DS macro code.

IF, THEN, and ELSE are reserved keywords; do not use them as variable names.

Syntax



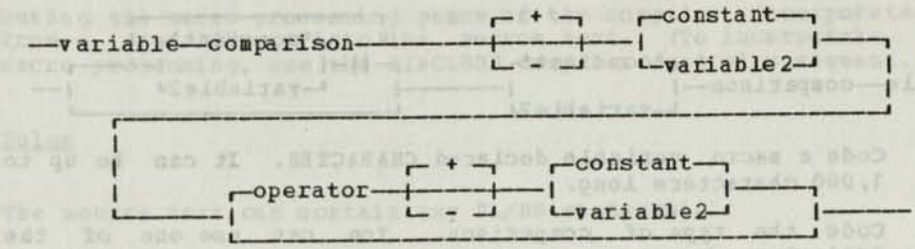
Operands

label	Optional. Code one or more labels, each followed by a colon.
relational expression	Code the %IF statement condition, a comparision of fixed or character values, described below.
%THEN%clause	Code the keyword %THEN and the action to be performed if the comparison is true. The action can be any macro statement except %MACRO or %END.
%ELSE%clause	Code the keyword %ELSE and action to be performed if the comparison is false. The action can be any macro statement except %MACRO or %END.

Note: You can code a null statement as the %THEN or %ELSE clause when there is no action to perform on that particular result of the IF comparison. (A %GOTO to a null statement at this point passes control to the lines following the null statement.)

Format of FIXED Variable Comparisons

When you want to compare a FIXED macro variable with another FIXED macro variable, a constant, or an expression, the format of the relational expression in the macro %IF statement is.



variable Code a macro variable declared FIXED.

comparison Code the type of comparison. You can use one of the following:

Operator	Meaning
=	equal to
¬=	not equal to
>	greater than
<	less than
>=	greater than or equal to
<=	less than or equal to
¬>	not greater than
¬<	not less than

Evaluation of a complex expression is by priority of the operators. The prefix operators are evaluated first, then *, /, // operators, and then + and - operators. If you want to change the order of evaluation, you may use parentheses. Operators and their associated operands enclosed in parentheses are evaluated first and are evaluated by priority.

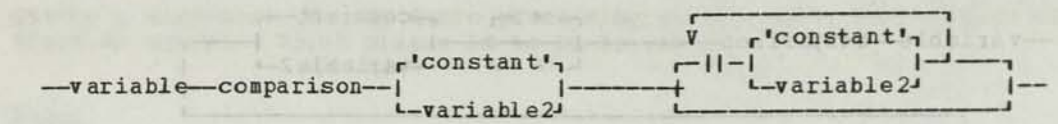
constant You must use a decimal number.

variable2 Code a macro variable declared FIXED.

operator Code the operation to perform on the values before the comparison is made. Use one of + - * / //.

Format of CHARACTER Variable Comparisons

When you want to compare a CHARACTER macro variable with another CHARACTER macro variable, a character constant, or an expression, the format of the relational expression in the macro %IF statement is:



variable — Code a macro variable declared CHARACTER. It can be up to 1,000 characters long.

comparison — Code the type of comparison. You can use one of the following:

Operator	Meaning
=	equal to
¬=	not equal to
>	greater than
<	less than
>=	greater than or equal to
<=	less than or equal to
¬>	not greater than
¬<	not less than

constant — Code a character string enclosed in parentheses. It does not have to be the same length as the macro variable. It can be up to 1,000 characters long.

variable 2 — Code a macro variable declared CHARACTER. It does not have to be the same length as the macro variable. It can be up to 1,000 characters long.

|| — You can concatenate character constants and macro variables with other character constants and macro variables using the concatenation operator (||). This operator simply connects the operands together forming a string with a length equal to the sum of the lengths of the operands. The total length can be up to 1,000 characters.

Note: If the macro variable and the value it is compared to are not the same length, the comparison adds blanks on the right of the shorter value, to make it equal in length to the longer value.

%INCLUDE -- Include Source Code from a User Library

Purpose

During the macro processing phase of the compiler, incorporates the text from a user library into the source text. (To incorporate text after macro processing, use the @INCLUDE compiler control statement.)

Rules

The source text can contain any PL/DS statements.

The included text is scanned during the macro phase in the same manner as the source text, with replacements being made and macro statements being executed. When all of the incorporated text has been scanned, scanning continues following the macro %INCLUDE statement.

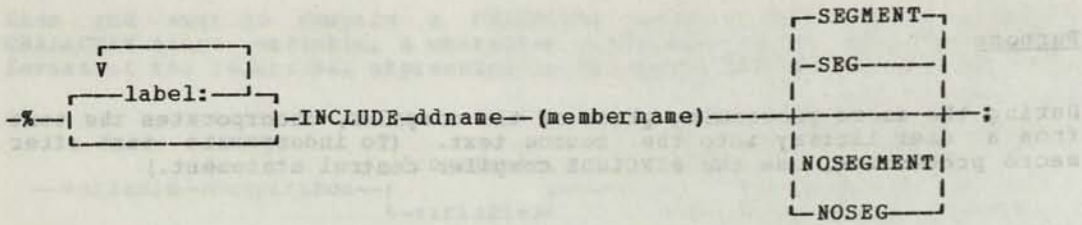
The included text that remains after macro processing stays in the source text for subsequent processing by the compile phase of the compiler.

The included text can contain %INCLUDEs and subsequent incorporated text can contain %INCLUDEs. The compiler accepts up to 14 levels of nested included code.

Segments created with @CREATE and @ENDCREATE at the first of the source data set can be included, using %INCLUDE, later in the same source data set.

Macro %INCLUDE statements appear as PL/DS comments at the end of the source listing.

Syntax



Operands

label — Optional. Code one or more labels, each followed by a colon.

ddname — Specify the name of the DD statement, in this compiler run's JCL, for the data set that contains the included text. (Typically, this is SYSLIB.)

membername — Specify the member that contains the included text. (You will need a separate %INCLUDE for each partitioned data set member you include.)

SEGMENT/SEG — Code this to cause the text included by this statement to be listed on a separate page after all non-included text. (This is an aid to keep the non-included text compact and its logic flow clear, especially in structured code under the FORMAT compiler option.)

NOSEGMENT/NOSEG — Code this to cause the compiler to list the text included by this statement at the place where it reads this statement.

Segment/Nosegment Default: Whichever segmentation parameter is in effect for the SOURCE compiler control option.

%INCLUDE Parameterization

When standard declarations for control blocks are INCLUDED it is often useful to be able to change the name and/or attributes on the level 1. The following example shows how this can be achieved:

```
File 'DCB COPY':

        %ACT ZNAME,ZATTR;
        %IF ZNAME='' %THEN
           %ZNAME='DCB';                /* SET NAME DEFAULT          */
        %IF ZATTR='' %THEN
           %ZATTR='BDY(WORD) BASED';    /* SET ATTR DEFAULT          */
        DCL 1 ZNAME ZATTR,
              3 DCBF1 ..
                - - -
                - - -
              3 DCBFN ;
        %ZNAME='';
        %ZATTR='';
        %DEACT ZNAME,ZATTR;             /* RESET FOR OTHER INCLUDES */
```

1. Default invocation:

```
        %INCLUDE SYSLIB(DCB);
```

...would return:

```
        DCL 1 DCB BDY(WORD) BASED,
              3 DCBF1 ..
                - - -
                - - -
              3 DCBFN  ;
```

2. Overriding defaults:

```
        %ZNAME='DCBE';
        %ZATTR='(8) BDY(WORD) LOCAL EXTERNAL';
        %INCLUDE SYSLIB(DCB);
```

...would return:

```
        DCL 1 DCBE(8) BDY(WORD) LOCAL EXTERNAL,
              3 DCBF1 ..
                - - -
                - - -
              3 DCBFN  ;
```

The %Null Statement in Macro Outer Code

Purpose

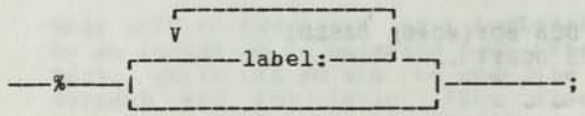
A statement that specifies a macro phase statement without any macro processing action to be performed.

Rules

When the target of a %GOTO statement is a null statement, processing proceeds with the statement following the null statement. A null statement is required when the target of a %GOTO statement is a macro invocation. See the %GOTO description.

When a %IF statement has a null statement as the THEN or ELSE clause, no action occurs when the comparison in the %IF statement leads to the null statement.

Syntax



Operand

label Optional. Code one or more labels, each followed by a colon. (A label is required if the null statement is the target of a GOTO or %GOTO statement.)

INVOKING MACRO DEFINITIONS

SUMMARY

You use a macro invocation to cause execution of the statements of a specified macro definition, during the macro processing phase of the compiler. Any answer text produced by the macro definition appears in the source code at the point where the macro invocation took place. The answer text is all that remains of all the macro source code after the macro processing phase. The answer text and compile source code (with target variables altered to their macro variable assigned values) are the input stream to the compile phase of the compiler.

You usually code a macro invocation by itself as a statement; however, you can also code it as the operand of a source statement.

If the value of a replaced target variable contains a macro invocation, the macro processing phase does not invoke the macro definition.

The definition of the macro you invoke must either be in a library, or must have been processed in your macro code ahead of your invocation of it.

If a macro with a certain name resides in a library and a macro with the same name is defined in the source program where that macro name is invoked, the macro definition in the program and not the library, is used.

The author of the macro must inform you how to use the macro, including the appropriate values to code in the invocation arguments.

For the PL/DS macro descriptions, see the publication, Distributed Programming Development System (DPDS), PL/DS Macros For DPPX Base, Reference.

PARAMETER PASSING

When a macro definition has provision for parameter passing, you include, in the invocation, a macro list to pass a positional argument list, and/or a keyname list to pass keyname argument lists.

Macro statements in the macro definition check to see what information you include on the macro invocation. In many cases the macro definition will supply a default value for any argument you do not specify in the macro invocation and for any keyname or keyname argument you did not specify. In order to invoke a macro definition you must know the macro name and be aware of the characteristics of the associated macro list, including any default values.

TARGET STRINGS IN MACRO INVOCATIONS

If a macro invocation's argument contains any target string that matches the name of a variable that is activated in the invoking environment, the macro processing phase of the compiler replaces the invocation variable with its presently-assigned variable value before executing the invoked macro.

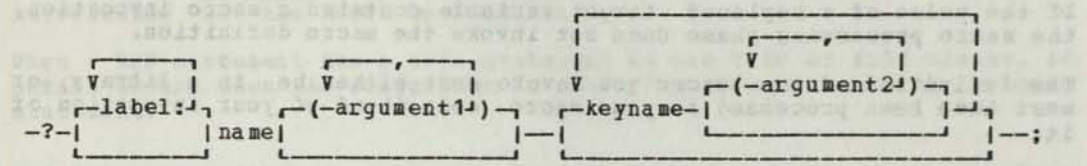
BLANKS AND COMMENTS IN MACRO INVOCATIONS

Before execution of the invoked macro, the macro processing phase removes blanks and comments from the macro invocation string. There are two exceptions.

The first exception is: any blank or comment that is part of a string constant is not removed.

The second exception is for any of the following: a blank, a succession of one or more blanks, or a comment. Any of these that is preceded or followed by an alphabetic character, numeric character, underscore, or apostrophe becomes a single blank before execution of the invoked macro.

Syntax



Operands

label — Optional. Specify one or more labels, each followed by a colon.

name — Code the name of the macro you are invoking. The name of a macro is specified in the %MACRO statement at the first of the macro definition.

argument1 — This is your positional parameter list of arguments for use by the macro. The arguments must be in the sequence established in the macro's definition.

Separate them by commas. To omit a particular positional parameter, code a comma in its place.

keyname — Code the name of one of this macro's keyname parameters, specified in the %MACRO statement at the first of the macro definition. You can code the keyname parameters in a different sequence than they appear in on the %MACRO statement.

argument2 — For each keyname, you can code either no parameters, one parameter, or a parameter list; depending on what is defined and appropriate for the macro you are invoking and the use you are making of it.

Examples of Invocations

```
?    MACRO12 (12,LIST,,NO) KEY1('010'B) KEY2(A,B,C);
```

...MACRO12 is the name of the macro. This invocation includes both a positional parameter list and keynames with their associated arguments.

```
A = B + ?X;
```

...X is the name of the macro in this source code assignment statement.

```
DO I = 1 TO 20 BY ?INCRMNT;
```

...INCRMNT is the name of the macro invoked in this source code DO statement.

The following are DPPX macro invocations:

```
?ASSIGNQ QID(JABP1) THR('25'X);

?XCTL ID(2) NAMEA(4) RNAME(TARGETA);
```

CHAPTER 5. CODING PL/DS MACRO DEFINITIONS

This chapter assumes that you are familiar with macro processing and invocations as described in the previous chapter.

A macro definition is a sequence of macro definition instructions that the macro processing phase initially processes and then executes wherever that macro is invoked in the user's source text. Macro definitions can perform two functions:

- They can assign values to variables, such as replacement values for target strings in non-macro text.
- They can specify source code statements for insertion into the source text for the compiler run. Source statements inserted by macro code are called "answer text."

Macro definitions can appear in the source program or can reside as members of a user's partitioned data set (DDNAME = SYSLIB). If a macro definition is included in the source program, only that source program may invoke the macro. A macro definition in a library can be invoked by any number of PL/DS programs.

THE RESPONSIBILITIES OF A MACRO DEFINITION PROGRAMMER

When you are producing macro definitions that will be invoked by many users, you are responsible for making them as readily usable as possible, and for supplying clear, complete instructions for their use. You can see an approach to such documentation in the publication, Distributed Processing Development System (DPDS), PL/DS Macros for DPPX Base: Reference.

Some of the things you should specify in a user description of a macro are:

- Which of the macro's several names (if more than one) to use for a certain function.
- The meanings of and appropriate values for both positional parameters and keyword parameters, including substringing information.
- The macro outer variables that the user must declare EXTERNAL (shared variables), and the significance of the values the macro returns in them.
- The value(s) -- if any -- that the user must submit in the MACPARM compiler option, for use by this macro.

These and other considerations will come to mind, if you put yourself in the shoes of the coder who will invoke your macro. Tell him enough, but do not tell him so much that he will be confused!

WRITING A MACRO DEFINITION

Writing a macro definition is very similar to writing a PL/DS procedure. Every macro definition has three parts:

1. A %MACRO statement first.

2. One or more macro definition statements.

3. A %END statement last.

The %MACRO statement begins the macro definition as the PROCEDURE statement begins a source code procedure. It is sometimes referred to as a header statement. You must code at least one name in the statement. This becomes the macro name used to invoke the macro.

The macro definition statements between the %MACRO and the %END statements can be any of the following:

```
ACTIVATE statement
ANSWER statement
Assignment statement
DEACTIVATE statement
DECLARE statement
DO statement
END statement
GOTO statement
IF statement
Null statement
RETURN statement
```

The %END statement defines the end of the macro definition.

You will notice that the names of some macro definition statements and macro outer statements are identical. There is usually a difference in their uses, however, and the descriptions in this chapter of the manual point out the differences.

SHARING VARIABLES BETWEEN MACRO DEFINITIONS

Each macro definition is a separate "environment" from any other macro definition and from the macro outer (non-definition statements) environment.

To share a variable among several environments in the same user program, you must declare that variable as EXTERNAL in each of those environments. If the user who is invoking your macros must make use of a macro definition variable you declare as EXTERNAL, he must be told to declare the variable EXTERNAL in his outer code.

One use of this would be for the macro definition to assign the value of a shared macro variable so that its corresponding non-macro target string would be replaced with an appropriate value corresponding to an invoked macro-definition-detected condition.

Note that a shared variable carries its most recent activated or deactivated status across environments.

MACRO DEFINITION FUNCTIONS

The macro definition functions are special functions that you can use in macro definition statements. They provide you with data that is not available with normal operators and operands.

When you code one of these functions in a macro definition statement, the compiler develops and uses the value it stands for in its place.

The 21 macro definition functions are all explained under "Using Macro Definition Functions."

The invocation data macro definition functions and the MACPARM function are especially useful for building macro logic based upon values and passed by invokers and compile time users, respectively.

MACRO DEFINITION ENVIRONMENT KEYWORDS

Figure 22 lists all of the words recognized by the compiler macro phase processing as macro definition environment keywords. Those keywords that are indicated as reserved cannot be used for variable names.

(A macro definition function is not a reserved keyword, but it cannot be used as both a macro definition function and variable name. Unless you explicitly declare it as user data, the macro definition function is categorized for the whole compiler run by the way you have used it the first time the compiler sees it.)

KEYWORD	ABBREVIATION	RESERVED
ACTIVATE	ACT	Yes
ANSWER	ANS	Yes
CHARACTER	CHAR	
CODE		
COLUMN	COL	
DEACTIVATE	DEACT	Yes
DECLARE	DCL	Yes
DO		Yes
ELSE		Yes
END		Yes
EXTERNAL	EXT	
FIXED		
GOTO	GO TO	Yes
IF		Yes
INTERNAL	INT	
KEYS		
MACRO		Yes
MESSAGE	MSG	
NORESCAN		
PAGE		
RESCAN		
RETURN		Yes
SKIP		
THEN		Yes

Figure 22. Macro Definition Keywords

COMPILER CONTROLS FOR MACRO PROCESSING

MACRO SOURCE CODE MARGINS

The macro definition statements in your source data set must observe the left and right source columns specified in the MARGINS compiler option. If there is any non-macro text outside of these columns, the compiler copies it to the output text of the macro processing phase. See "Compiler Options."

COMPILER CONTROL STATEMENTS

You may code @SPACE and @EJECT compiler control statements within a macro definition.

THE FOUR STEPS OF MACRO USE

This section gives an example of a requirement, and explains the steps involved in constructing and invoking a PL/DS macro to meet that requirement.

In this example, a PL/DS macro ?LINK is required so that the invocation:

```
DCL R18   PTR REG(18);
DCL R16 FIXED(16) REG(16);
DCL ENTP  PTR;
DCL ENT   ENTRY BASED(ENTP) VALUERANGE(*);
?LINK EP(ENT);
```

...will generate:

```
DO;
  RFY (R16, R18) RSTD;
  R18=ADDR(ENT);
  R16=5678;
  KIZ(127);
  RFY (R16, R18) UNRSTD;
END;
```

The following steps are required to achieve this:

Step 1: Create the macro.

The following macro definition must be created:

```
%LINK:MACRO KEYS(EP);
    ANS('DO;')                          COL(MACCOL);
      ANS('RFY (R16, R18) RSTD;')       COL(MACCOL+2) SKIP;
      ANS('R18=ADDR('||EP(1)||');')     COL(MACCOL+2) SKIP;
      ANS('R16=5678;')                  COL(MACCOL+2) SKIP;
      ANS('KIZ(127);')                  COL(MACCOL+2) SKIP;
      ANS('RFY (R16, R18) UNRSTD;')     COL(MACCOL+2) SKIP;
    ANS('END;')                         COL(MACCOL) SKIP;
%END;
```

Step 2: Catalog the file containing the macro definition on a partitioned data set.

Step 3: Add the macro invocation to the source program.

For example:

```
T:PROC;
  - - -
  %INCLUDE SYSLIB(LINK); /* NOT ESSENTIAL - BUT USEFUL
                            WHEN AND ONLY WHEN
                            DEBUGGING THE MACRO      */
  - - -
  ?LINK EP(FRED);        /* INVOKE LINK MACRO        */
  - - -
END;
```

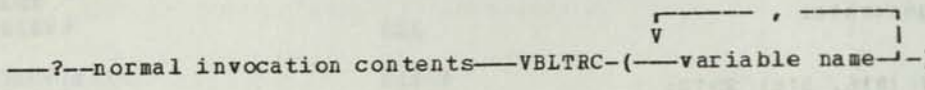
Step 4: Compile the source program.

Whenever the invocation, ?LINK, appears in the source program, the specified code will be prepared for the compile phase.

MACRO DEFINITION DEBUGGING AIDS

When debugging ?macros, use the MACRO and MSOURCE options. In addition, the VBLTRC option lists each change in value of any macro variable that you specify. The variable value changes are printed on the macro phase listing at the location of the invocation.

You code the option as a keyword preceding the semicolon at the end of your test run invocation as follows:



...where "variable" is the name of the a macro definition variable whose value changes during macro execution are to be listed. In the following example:

```
?LINK EP(ENT1) VBLTRC(JOE,SAM);
```

...the changes to variables JOE and SAM are printed during execution of the macro LINK.

SUMMARY OF MACRO DEFINITION STATEMENTS

The macro definition statements are the part of the PL/DS macro language that you choose from when you define a macro. You use them to:

- Describe the logic of the macro.
- Produce ANSWER text.
- Set macro definition variables.

Each macro definition consists of: a %MACRO statement, in which you specify the macro name and keyword names; the set of macro definition statements that specify the action or set of actions that are to occur when you invoke this macro; and the %END statement, that signifies the end of this macro.

Within the macro definition statements you may code macro definition functions, a special set of functions described under "Coding Macro Definition Functions." Figure 23 lists the macro definition statements covered in this section.

STATEMENT	USE
ACTIVATE	Causes the specified macro definition variable's assigned value to replace eligible matching target strings.
ANSWER	Contains an expression that is evaluated and made part of the source text to be compiled.
assignment	Assigns a value to a macro definition variable.
DEACTIVATE	Blocks the specified macro definition variable's assigned value from replacing eligible matching target strings.
DECLARE	Defines the data type and scope of a macro variable.
DO	Begins a unit of code within a macro definition.
END	Ends a unit of code within a macro definition.
%END	Marks the end of a macro definition.
GOTO	Unconditional branching within a macro definition.
IF	Provides conditional macro statement execution.
%MACRO	Marks the beginning of a macro definition.
null	Causes no action.
RETURN	Returns control to the invoker of a macro.

Figure 23. Table of Macro Definition Statements

ACTIVATE -- Activate Macro Definition Variable

Purpose

You use the macro definition ACTIVATE statement to restore a previously deactivated macro definition variable to "activated" status.

(Note that there is a macro outer %ACTIVATE statement, for macro outer target strings.)

The target strings that are eligible to be replaced by the assigned value of activated macro definition variables are explained in the description of the Assignment statement.

Rules

The macro processing phase uses the assigned value of an activated macro definition variable to replace, in each subsequent macro definition ANSWER statement that it executes in this macro definition, each eligible target string that matches the macro definition variable's name.

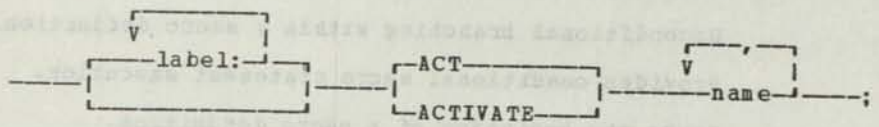
The default state of each macro definition variable declared INTERNAL is activated.

The default state of each macro definition variable declared EXTERNAL is activated, unless the most recent environment that declared the same variable name EXTERNAL has also deactivated it.

Answer text is rescanned for target strings except for either (a) the macro variable values you earlier deactivate with macro definition DEACTIVATE statements, or (b) answer text in whose ANSWER statement you specify NORESCAN. The DEACTIVATE statement in (a) above would either appear earlier in the present macro definition, or, for a variable declared EXTERNAL, could appear in a macro definition that also declares it EXTERNAL and is invoked prior to the invocation of the present macro definition.

During the macro processing phase of the compiler, ACT and ACTIVATE are reserved keywords; do not use them as variable names.

Syntax



Operands

label Optional. Code one or more labels, each followed by a colon.

name Code the target variable's name: a character string in answer text to become eligible for replacement by its corresponding macro variable's assigned value.

ANSWER -- Generate Source Text, Invoke Inner Macro, Print Message

Purpose

You use the macro definition ANSWER statement to specify a line of compiler source code to be inserted in this location during the macro processing phase of the compiler. You can use the ANSWER statement to invoke another macro from within the macro definition. Also, you can specify message text to be printed on the MSOURCE listing.

Rules

The ANSWER statement is the only way to invoke another macro within a macro. You can invoke a further macro from invoked macros for a total of 15 levels of macros including the first one.

The PL/DS MACGEN compile statement is valid only when it is answer text from an ANSWER statement in a macro definition that is invoked from a macro library.

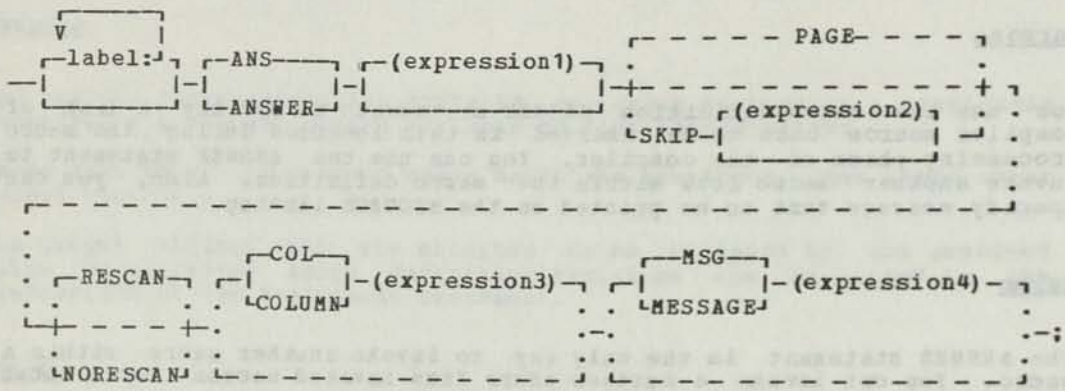
Answer expression text is scanned and eligible target strings are replaced using a different procedure from that used for other eligible target strings during the macro phase of the compiler. For a description of processing of ANSWER statements, see, "The Sequence of Processing ANSWER Statements," below.

You can code a macro definition that contains none or one or more ANSWER statements.

The macro processing phase positions the resulting answer text or message at the place in the source code where it processes the ANSWER statement that specified the answer or message text (see your MSOURCE listing).

During the macro processing phase of the compiler, ANS and ANSWER are reserved keywords; do not use them as variable names.

Syntax



Operands

label — Optional. Code one or more labels, each followed by a colon.

expression1 — Code, in parentheses, the expression with which the macro processing phase of the compiler will produce answer text. The variables, macro definition functions, and constants that you code in the expression must all be CHARACTER.

You may code a macro invocation, provided it is at the end of the answer expression. See under "Rules" above, in this description.

See, "Coding the Answer Expression," below.

SKIP(expression2) — Specifies that the answer text produced from expression1 is to be on the new source listing line. If you do not wish the default skip of 1 (next line), code a fixed expression. (SKIP, like the compiler control word, SPACE, does not space any further than the top line on the next page.)

If you code both SKIP and PAGE, SKIP is ignored.

PAGE — Specifies that the answer text produced from expression1 is to be at the top of a new page. (PAGE is like the compiler control word, EJECT.)

If you code both SKIP and PAGE, SKIP is ignored.

RESCAN — Specifies that the answer text expression is to be scanned for target variables that correspond to macro variables, and the variables replaced by their macro variable values. RESCAN is the default if you do not code NORESCAN. See "Target Variables in Answer Text," below.

NORESCAN — Specifies that you do not want scanning of this answer expression, with replacement of target strings, to take place. NORESCAN is in effect only for each single ANSWER statement you code it in.

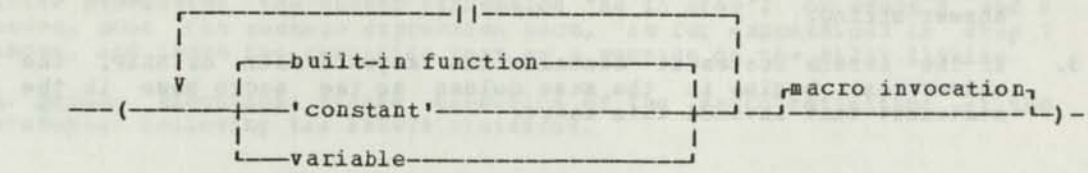
COL/COLUMN (expression3)
: Specifies the column of the source program listing that the answer text is to start in. (See the MACCOL macro definition function for a useful example.) Code an expression that is either a fixed constant or a fixed variable or a complex arithmetic expression. You may use any macro definition functions, so long as the result is arithmetic. This parameter is in effect for this statement only and on the source listing only, even if the compiler option FORMAT COMPILE is in effect.

  See also, "Default Values for the COLUMN Keyword," below.

MSG/MESSAGE (expression4)
: Specifies a message, in expression4, to be printed on the msource listing at the place where the macro containing this ANSWER statement was invoked. Use only character variables and constants in the expression4 message text. (On the source listing, the ANSWER statement indicates the first of any answer text.)

Coding the Answer Expression

The answer expression (expression1, above) has the following syntax:



built-in function
: Code a function that returns (resolves to) a character string.

'constant'
: Code a character string, enclosed in apostrophes.

variable
: Code a character variable (subscripted, if in array).

macro invocation
: Optional. Code a macro invocation at the very end of the ANSWER expression.

The answer text produced from the expression in the ANSWER statement can be any character string. This includes assembler language statements and all PL/DS statements other than macro statements (the macro invocation is an exception). Answer text that is to be compiled must be PL/DS code, or it must combine with surrounding strings to make PL/DS code. (Reserved words that appear in answer text must conform to PL/DS rules for their use.)

Default Values of the COLUMN Keyword

The COLUMN keyword indicates that the starting column for printing the answer text. Be certain to specify a column greater than MACLMAR and less than or equal to the right source margin, specified in the MARGINS compiler option. (You can obtain this value with the MACRMAR, macro definition function.)

If you specify a column number for a line and that column already contains answer text (from a preceding ANSWER statement), the new ANSWER statement's printed answer text will begin on a new line.

If the answer text does not all fit on the line, part of it will be placed up to the right source margin (MACRMAR); the remainder is continued on the next line at the left source margin (MACLMAR).

If you omit the COLUMN Keyword, one of the following three things occurs:

1. If this is the first ANSWER statement in the macro definition, the answer text begins in the same column as the macro name in the macro invocation.

2. If an ANSWER statement precedes this ANSWER statement, and PAGE or SKIP is not specified or implied, the answer text begins in the column immediately following the last character of the previous answer string.

3. If the ANSWER statement contains the keyword PAGE or SKIP, the answer text begins in the same column as the macro name in the statement that invoked this macro.

The Sequence of Processing ANSWER Statements

When the macro processing phase of PL/DS executes an ANSWER statement as part of an invoked macro definition, there is a set sequence of operations performed on the contents of the ANSWER statement. The sequence is as follows:

1. Scan Answer Expression (expression1 in ANSWER Syntax): Regardless of whether or not NORESCAN is specified, scan the whole answer expression once, without recognizing any inner macro invocation that it contains. Replace any macro definition functions (including any macro invocation positional and keyword parameters) with their character values. Replace any eligible target variable with its immediate macro definition variable's assigned value, but do not rescan the replaced string.

Macro definition built-in functions are recognizable when coded as individual tokens separated by the concatenation operator (||). (No enclosing single quotes are allowed. For example, MACNAME is recognized and replaced, but 'MACNAME' is not.)

2. Produce Answer Text (NORESCAN In Effect): If NORESCAN is specified, place the entire answer expression that results from step 1 into the output stream of the macro processing phase, obeying the PAGE, SKIP, or COL that is specified in the ANSWER statement.

(If the answer text contains an inner macro invocation, the inner macro is not executed, and because it remains in the output it subsequently causes a compile phase error.)

3. Produce Answer Text (RESCAN in Effect): If RESCAN is in effect (either defaulted or RESCAN specified) rescan the ANSWER expression that results from step 1 for target strings that match activated macro variables, and replace each such target string with the value presently assigned to its matching macro definition variable. Immediately rescan and re-rescan each such replaced token until no further replacements are found for it, then continue scanning the answer expression for another eligible target string.

Place the resulting text, up to any answer expression inner macro invocation, into the macro processing phase output data set, obeying the PAGE, SKIP, or COL that is specified in the ANSWER statement.

4. Execute Inner Macro Invocation (RESCAN in Effect): If an inner macro invocation within the answer expression is encountered while rescanning, rescan any eligible target strings it contains and execute that macro definition immediately.

(A macro invocation that is contained in the replacement value of an answer expression target string is executed because the invocation trigger character (?) is recognized during the re-rescan.)

Process any answer expression in the inner macro in the same way as described in this section.

5. Issue Message Text (expression4 in ANSWER Syntax):

After processing the answer expression as in step 2 or steps 3 and 4 above, scan the message expression once, as for expression1 in step 1 above, and issue the resulting text as a message on the MLIST listing.

6. Resume Execution: Resume execution of the macro definition at the statement following the ANSWER statement.

Avoiding Messages That are Out of Sequence With Answer Text

For a series of successively nested inner macros invocations, their MLIST messages may appear in inverse order to their answer text and invocations.

This occurs when MESSAGE is coded in ANSWER statements whose expression1 contains an inner macro invocation: The successive macros, ANSWER statements, and inner invocations in expression1 are processed and executed, until no further nested invocation occurs; the innermost nested macro is completed and processing returns to its invoking ANSWER statement; that ANSWER statement's message is processed; upon completion of its macro, processing "backs out" to the next-to-innermost invoking ANSWER statement and issues its message, and so on.

To have messages appear in the same sequence as their corresponding answer text and macro invocations, code the ANSWER statement with MESSAGE (expression4) only, then code the ANSWER statement with expression1 only.

Example of ANSWER Macro Sub-Invocations With Parameter Passing

The following are three macro definitions, M1, M2, and M3. They illustrate, in ANSWER statements containing macro sub-invocations, the uses both of literal keynames that don't get substituted and of keynames that do get substituted. (See the description of the macro definition Keyname function.)

The author of M1 called for three invocation keyname parameters:

```
% M1: MACRO KEYS (AB, CD, EF);
 - - -
 - - -
% END;
```

The author of M2 has found a use for M1 and developed logic to satisfy keyname EF, leaving only two keyname parameters for the invoker of M2 to specify:

```
% M2: MACRO KEYS (AB,CD);
 - - -
  DCL VAREF. . .;
  VAREF = 'LOU'||CD(1);
 - - -
 - - -
  INVOKM1:ANSWER
       ('?M1'||' AB'||AB||' CD'||CD||' EF('||VAREF||');');
 - - -
 - - -
% END;
```

The author of M3 builds upon the M2 macro. He builds logic to generate a value for keyname CD, leaving only AB for the M3 invoker to specify:

```
% M3: MACRO KEYS (AB);
 - - -
  DCL VARCD. . .;
  VARCD = AB(1)||'1';
 - - -
  INVOKM2: ANSWER ('?M2'||' AB'||AB||' CD('||VARCD||');');
 - - -
 - - -
% END;
```

Given this simple invocation:

```
? INVOKM3: M3 AB(5491);
```

...statement INVOKEM2 generates this sub-invocation:

```
?M2 AB(5491)CD(54911);
```

...and subsequently statement INVOKEM1 produces this second level inner invocation:

```
?M1 AB(5491)CD(54911)EF(LOU54911);
```

In this way, then, you can see how you can make use of and sometimes simplify existing macros by defining outer macros that build the required parameter values, and then pass these values to their appropriate keynames in internal ANSWER expression invocations.

Examples of the Macro Definition ANSWER Statement

```
%X:       MACRO;
          DCL(A,B)CHAR;
          A='VBL';
          B='+';

          ANS('A=A+B+'||CHAR(LENGTH(A))||';');
                    /*  RETURNS TO THE COMPILER: */
                    /*  VBL=VBL+++3;             */

          ANS('A=A+B;')NORESCAN;
                    /*  RETURNS TO THE COMPILER: */
                    /*  A=A+B;                   */

/*    MESSAGE EXPRESSIONS */

ANS   MSG('A CONTAINS AN INVALID VALUE');
                    /*  THE MESSAGE ON THE */
                    /*  SOURCE LISTING IS: */

/*    MSG VBL CONTAINS AN INVALID VALUE */

ANS   MSG('A CONTAINS AN INVALID VALUE') NORESCAN;
                    /*  THE MESSAGE ON THE */
                    /*  SOURCE LISTING IS: */
                    /*  MSG A CONTAINS AN INVALID VALUE  */

/*    NESTING MACRO INVOCATIONS    */

          ANS('A=?Y;');
                    /*  RETURNS TO THE COMPILER: */
                    /*  VBL=10;                  */

%END      X;


%Y: MACRO;
          - - -
          - - -
          - - -
          ANS('10;')
                    /*  THIS ANSWER IS RETURNED        */
                    /*  TO THE MACRO X WHICH           */
                    /*  INVOKES Y                      */
%END      Y;
```

Assignment -- Specify Macro Definition Variable Value

Purpose

You use an assignment statement to assign a value to a macro definition variable. You can use any of the macro definition functions in an assignment statement.

Rules

The value or values (source) you assign must be the same data type as you declared for the macro variable (receiver).

References to positional arguments and keyname arguments are treated as CHARACTER variables. A subscript expression can follow any variable declared as an array; a substring expression can follow any CHARACTER variable. How to code subscript and substring expressions is described under "Subscripts and Substrings in Macro Definition Assignments," below.

Target String Replacement by Macro Definition Variable Values

For a macro definition variable declared INTERNAL, the only matching target strings that its assigned value will replace during execution must be processed after its assignment, and are: (a) those target strings in answer text that this same macro definition or its sub-invoked macros produces, and (b) those target strings in the parameter lists of subinvocations by this macro.

In order to make a macro definition variable's assigned value replace target strings in the macro outer environment (including non-macro text), you must declare that macro variable as EXTERNAL and ACTIVATE it, and you must instruct the programmer who invokes this macro to declare a macro outer variable (%DECLARE), with the same name, as EXTERNAL and he must ACTIVATE it.

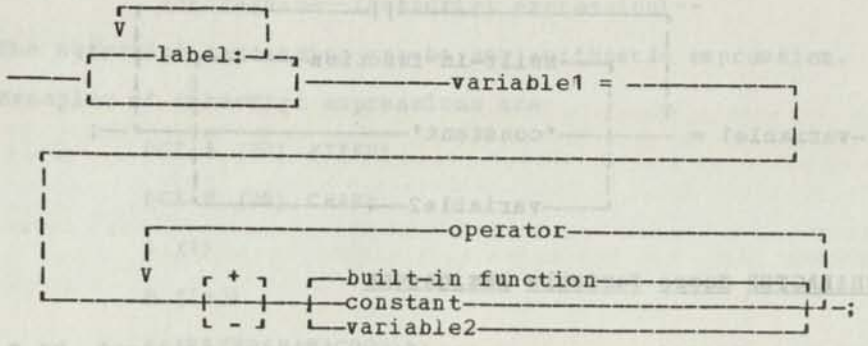
A target string is not eligible for replacement if it is in comments (/* */) or in string constants, or has a letter, digit, or apostrophe either just before or just after it.

The value that you assign to a macro variable should not contain unmatched comment delimiters (/* */) or unmatched string delimiters (' '), unless the target variable that it replaces is concatenated (%%) to another target variable that will have the matching delimiters. An unmatched opening delimiter causes the compiler to continue scanning in the search for a closing delimiter.

The only macro statement that a character constant in an assigned value can contain is a macro invocation.

The character constant can be a null string (e.g. B = '';), in which case the receiver is assigned a length of zero. Apostrophes enclosing character constants do not appear in the value assigned to the receiver, nor are they counted in the length.

Syntax for Assigning a FIXED Macro Variable



Operands for FIXED Macro Variable Assignment

label — Optional. Code one or more or more labels, each followed by a colon.

variable1 — Specify the FIXED macro variable (receiver) that is to have assigned to it the value (source) you are coding after the equal sign. You can use subscripting only if the variable was declared with a dimension.

= — The equal sign identifies this as an assignment statement. The source expression follows the equal sign.

constant — Must be a decimal number in the range -2147483647 to +2147483647.

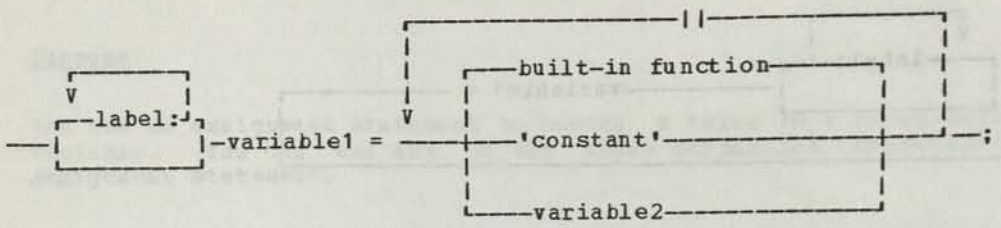
variable2 — Another macro variable declared FIXED, that has an assigned value. (You can code variable1 here, when its present value is involved in the computation of its new value.) You can use subscripting only if the variable was declared with a dimension.

operator — Specify the operation to be performed to compute the value of the receiver.

Use one of: + - * / // (<-remainder)

Evaluation of a source expression that contains operators is done by priority of the operators. Built-in functions are evaluated first, then prefix operators, then *, /, // operators, and then + and - operators. If you want to change the order of evaluation, you may use parentheses. Operators and their associated operands enclosed in parentheses are evaluated first and are evaluated by priority.

Syntax for Assigning a CHARACTER Macro Variable



Operands for CHARACTER Macro Variable Assignment

label — Optional. Code one or more labels, each followed by a colon.

variable1 — Specify the CHARACTER macro variable (receiver) that is to have assigned to it the value (source) you are coding after the equal sign. You can use subscripting only if the variable was declared with a dimension. It may be substringed.

= — The equal sign identifies this as an assignment statement. The source expression follows the equal sign.

'constant' — Code a character string, enclosed in apostrophes.

variable2 — Another macro variable declared CHAR, that has an assigned value. (You can code variable1 here, when its present value is involved in the computation of its new value.) You can use subscripting only if the variable was declared with a dimension. It may be substringed.

|| — You can assign a string that contains a concatenated series of functions and/or character constants and/or macro variables. The maximum length allowed for the string you build is 1,000 bytes.

Notes:

1. Character constants, enclosed in apostrophes, and CHARACTER macro variables can appear in the source expression of the macro assignment statement. The receiver is assigned both the value and the length of the source expression. Apostrophes enclosing character constants do not appear in the value assigned to the receiver. The value assigned to the receiver should not contain unmatched comment or string delimiters.

2. A character constant source expression can contain a macro invocation but no other macro statements (the rescan will ignore them and the compile phase will treat them as errors). The character constant can be a null string (for example, B= '';) in which case the receiver is assigned a length of zero.

3. The value assigned to the receiver should not contain unmatched comment or string delimiters. However, the receiver may have unmatched comment or string delimiters if they are matched in another variable concatenated (||) to the receiver.

Subscripts and Substrings in Macro Definition Assignments

Both variable1 and variable2 in an assignment statement in a macro definition can contain subscript expressions. Variable2 can contain substring expressions, but variable1 may not. The rules for coding these expressions are described below.

A subscript expression is used to reference an element of an array; the expression, enclosed in parentheses, follows the array name:

```
--arrayname--(subscript expression)--
```

The subscript expression can be any arithmetic expression.

Examples of subscript expressions are:

```
DCL A (10) FIXED;

DCL B (20) CHAR;

A (1)

B (I+3)

A (FIXED(B(MACCOL)))
```

A substring expression is used to reference a portion (one or more characters) of a character string; the expression, enclosed in parentheses, follows the variable name assigned to the character string:

```
--variable name (substring expression)--
```

Substring expressions can only appear in the source expression.

Substring expressions can contain decimal constants, FIXED variables that are subscripted or unsubscripted, and any of the macro definition functions whose value is arithmetic. The operators that can appear in the expression are prefix plus, prefix minus, addition, subtraction, multiplication, division, and remainder operators (+, -, *, /, //). When the expression contains operators, parentheses can be used to change the order of evaluation. Operators and their associated operands enclosed in parentheses are evaluated first, and are evaluated by priority of the operators.

To reference one or more characters, you must indicate the position in the character string of the first character and the position of the last character. The first position and the last position are separated by a colon. The format of the first and last positions is coded as described in the previous paragraph. For a single position, code its number both before and after the colon. Note that this is more restrictive than compile phase source substring expressions, where you can omit the upper limit when you are referencing a single character.

When you reference more than one character, the range is checked:

1. The value of the first position must not be greater than the value of the last position.

2. The value of the first and last position must be in the range of the character string.

To reference a portion of a character string that is an element of an array, the substring expression is preceded by the subscript expression. The two expressions are separated by a comma.

Examples of substring expressions are:

```
DCL B(20) CHAR;

DCL C CHAR;

C(I:J)

C(FIXED(B(3)):FIXED(B(3))+4)

B(1,1:3)

B(I,J:K)
```

DEACTIVATE -- Deactivate Macro Definition Variable

Purpose

You use the macro definition DEACTIVATE statement to specify to the compiler that a presently activated macro definition variable is no longer eligible to have its assigned value placed into an eligible target string.

(Note that there is a macro outer %DEACTIVATE statement, for macro invocations, answer text, and non-macro text target variables.)

Rules

The default state of each macro definition variable declared INTERNAL is activated.

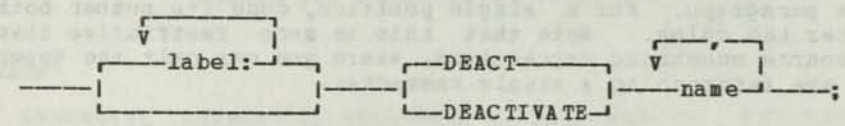
The default state of each macro definition variable declared EXTERNAL is activated, unless the most recent environment that declared the same variable name EXTERNAL has also deactivated it.

If a DEACTIVATE statement is not scanned, because of a macro GOTO statement, the variable remains activated.

Any declared variable that has been deactivated is still available for use in macro definition statements, and its value can be modified while it is deactivated. Arrays, labels, keynames, and built-in function names are always deactivated.

During the macro processing phase of the compiler, DEACT and DEACTIVATE are reserved keywords; do not use them as macro variables.

Syntax



Operands

label — Optional. Code one or more labels, each followed by a colon.

name — Specify the answer text target variable that is no longer to be replaced in answer text by its corresponding macro variable's assigned value.

DECLARE -- Declare a Macro Definition Variable

Purpose

You use the DECLARE statement in a macro definition to assign to a macro definition variable the attributes of FIXED or CHARACTER, and INTERNAL or EXTERNAL. (Note that there is also a macro outer %DECLARE statement, whose INTERNAL macro variable values are not available to executing macro definitions.)

Rules

You cannot assign a precision for FIXED macro variables or a length for CHARACTER macro variables. A FIXED macro variable is represented internally as a binary word with an initial value of zero. A CHARACTER variable is a variable length string. The length of the variable changes as values are assigned to it. The initial length of the variable is zero.

You cannot initialize the macro variable in this statement. Use the assignment statement.

The DECLARE statement for a macro variable must come before any use of that macro variable. Any macro variable that is not declared is assigned a data type of CHARACTER; the default scope attribute is INTERNAL.

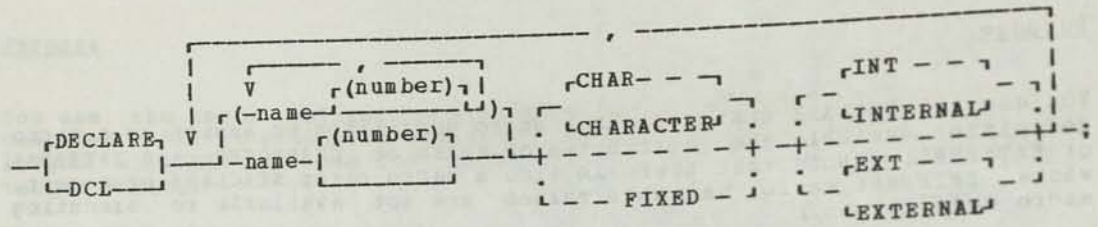
You can declare more than one macro variable on the same macro DECLARE statement. If a number of macro variables have the same attributes, separate the variables with commas and enclose the list in parentheses; follow the list with the attributes.

Macro variables declared INTERNAL in a macro definition DECLARE statement are known only in macros invoked within the macro that declares them. They are are not known outside the macro definition they are coded in, and they disappear once the macro definition where they were declared finishes executing.

A macro definition variable declared EXTERNAL in one macro and a macro definition variable of the same name declared EXTERNAL in any other macro, or a macro outer variable of the same name declared EXTERNAL, share (a) the value most recently assigned to that variable in macro phase processing, and (b) the most recently set activated/deactivated status.

DCL and DECLARE are reserved keywords; do not use them as macro variables.

Syntax



Operands

name — Code the name of the macro variable you are declaring.

The maximum length of the name is 16 characters. The first character must be alphabetic (A-Z). The remaining characters must be alphabetic (A-Z), numeric (0-9), or underscore (_).

(number) — You can declare a variable as an array by coding the number of elements in the array (maximum 255) in parentheses right after the variable name.

FIXED — The macro variable has the attribute, fixed.

CHARACTER/CHAR — The macro variable has the attribute, character.

INTERNAL/INT — The macro variable has the scope attribute, internal. This is the default scope attribute.

Internal macro variables are not known to any macro definition that is invoked during the macro phase.

EXTERNAL/EXT — The macro variable has the scope attribute, external. EXTERNAL macro variables can be referenced outside of and within a macro definition.

The first time the EXTERNAL variable is declared, the value of a FIXED variable is zero and the value of a CHARACTER variable is a null string. Thereafter, the value of the variable is the most recent value assigned to it; this value can be changed only by executing an assignment statement.

Declaring Multiple Macro Variables In One Statement

Factoring can be used in the DECLARE statement; however, only variable names and dimensions can appear in parentheses. For example:

```
DCL (A(10),B,C) FIXED INTERNAL;
```

You can declare more than one variable in a DECLARE statement without using factoring. For example:

```
DECLARE X FIXED, Y CHARACTER;
```

DO -- Start a Macro Definition Do-Group

Purpose

You use a DO statement in a macro definition to start a do-group: a group of statements that you can execute all together or bypass altogether.

Rules

You mark the end of a do-group with an END statement.

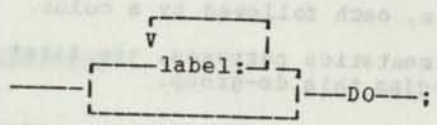
The statements that make up the do-group can be any of the statements that can appear in a macro definition except a DECLARE statement, a MACRO statement, or a %END statement for the macro definition.

Do-groups can be nested up to 15 levels.

Looping do-groups are not supported in PL/DS macro code.

DO is a reserved keyword; do not use it as a variable name.

Syntax



Operands

label Optional. Code one or more labels, each followed by a colon.

Example of a DO Statement

An example of using the DO statement is:

```
IF A = B THEN
     DO;
     - - -  ┐
     - - -  ├>- DO-GROUP FOR "YES" ANSWER
     - - -  ┘
     END;
ELSE
     DO;
     - - -  ┐
     - - -  ├>- DO-GROUP FOR "NO" ANSWER
     - - -  ┘
     END;
```

END -- End a Macro Definition Do-Group

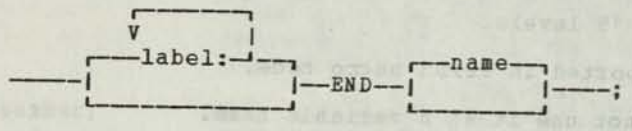
Purpose

You use an END statement to end a macro definition do-group, started with a macro definition DO statement. (Note that there is a %END statement, for ending a macro definition.)

Rules

END is a reserved keyword; do not use it as a variable name.

Syntax



Operands

label Optional. Code one or more labels, each followed by a colon.

name Optional. You can code, for documentation purposes, the first label on the DO statement that begins this do-group.

%END -- End a Macro Definition

Purpose

You use a %END statement to signify the end of a macro definition (see %MACRO).

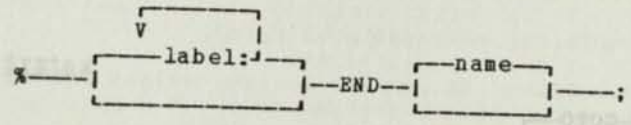
(Note that there is also a macro definition END statement, for ending a macro definition do-group.)

Rules

The %END statement causes a return of control to the caller of the macro definition, if processing reaches this statement.

END is a reserved keyword; do not use it as a variable name.

Syntax



Operands

label	Optional. Code one or more labels, each followed by a colon.
name	Optional. You may code the name that is on the %MACRO statement.

GOTO -- Transfer Control Inside a Macro Definition

Purpose

You use the macro definition GOTO statement to branch unconditionally to a labelled statement before or after it during execution of this macro definition.

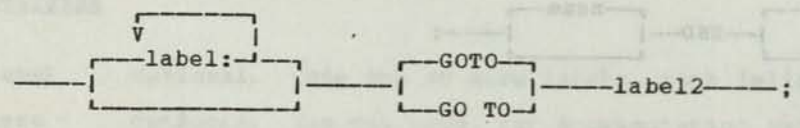
(Note that there is a macro outer %GOTO statement, for branching in macro outer code, and it cannot branch backwards).

Rules

The GOTO statement and the labelled statement it branches to must be in the same macro definition.

GOTO and GO and TO are reserved keywords; do not use them as variable names.

Syntax



Operands

label — Optional. Code one or more labels, each followed by a colon.

label2 — Code the label of the statement within this macro definition that you wish to branch to.

Examples of the GOTO Statement

Some examples of using the GOTO statement in a macro definition are:

```
% M1: MACRO;
  - - -
  - - -
LAB1: LAB2: A=B;
  - - -
  - - -
GOTO LAB2;
  - - -
  - - -
GOTO LAB3;
  - - -
  - - -
LAB3: IF A=B THEN ...;
  - - -
  - - -
%END;
```

IF, THEN, (ELSE) -- Conditional Macro Definition Statement Execution

Purpose

You use the IF statement in a macro definition to state a condition under which an accompanying THEN or optional ELSE clause is to be executed. (Note that the %IF, %THEN, and %ELSE, with the percent character, have identical uses in macro outer code, but do not allow the use of macro definition functions in the relational expression.)

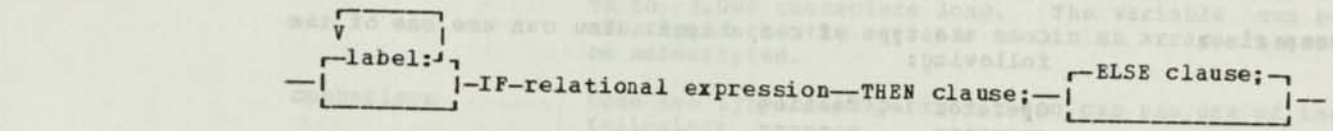
Rules

If there is no ELSE clause, and the IF condition is false, processing continues with the source text following the THEN clause or THEN do-group.

Multiple IF relational expressions, joined by | and &, are not supported in PL/DS macro code.

IF, THEN, and ELSE are reserved keywords; do not use them as variable names.

Syntax

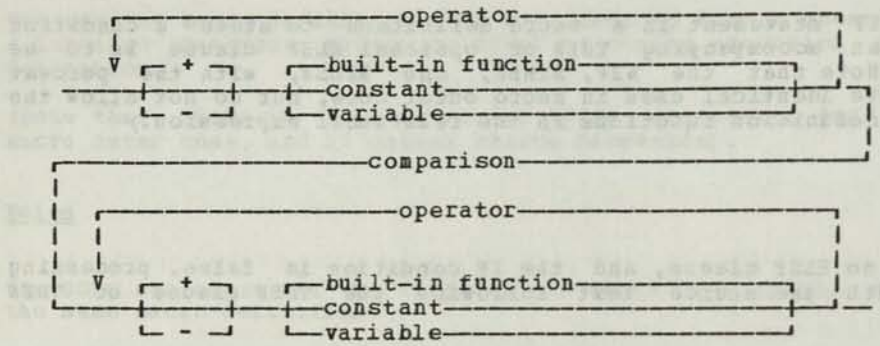


Operands

label	Optional. Code one or more labels, each followed by a colon.
relational expression	Code the IF statement condition, a comparison of fixed or character values, described below.
THEN clause	Code the keyword THEN and the statement or statements to be performed if the condition is true.
ELSE clause	Code the keyword ELSE and the statement or statements to be performed if the condition is false.

Format of Fixed Value Comparisons

When you want to compare a FIXED macro variable with another FIXED macro variable, a constant, or an expression, the format of the relational expression in the macro IF statement is:



built-in function — You may code a macro definition function if the value it results in is FIXED.

constant — You must code a decimal number.

variable — Code any macro variable declared FIXED.

operator — Code the operation to perform on the values before the comparison is made. Use one of + - * / // .

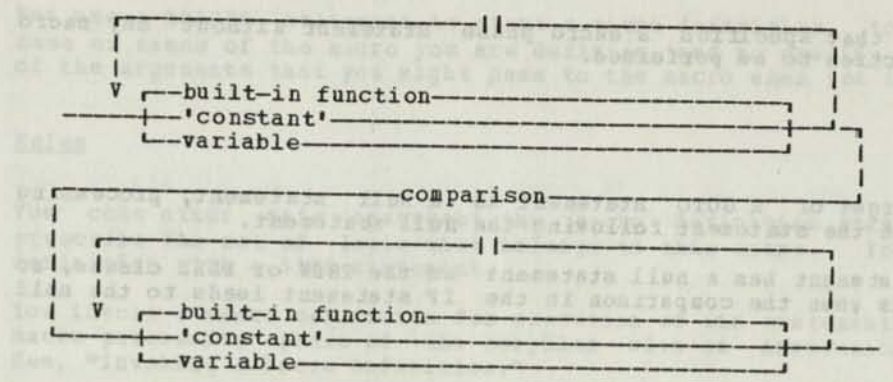
comparison — Code the type of comparison. You can use one of the following:

Operator	Meaning
=	equal to
¬=	not equal to
>	greater than
<	less than
>=	greater than or equal to
<=	less than or equal to
¬>	not greater than
¬<	not less than

Evaluation of a complex expression is by priority of the operators. Built-in functions are evaluated first, then prefix operators, then *, /, // operators, and then + and - operators. If you want to change the order of evaluation, you may use parentheses. Operators and their associated operands enclosed in parentheses are evaluated first and are evaluated by priority.

Format of Character Value Comparisons

When you want to compare a CHARACTER macro variable with another CHARACTER macro variable, a character constant, or an expression, the format of the relational expression in the macro IF statement is:



built-in function — You may code a macro definition function if the value it results in is CHARACTER.

'constant' — Code a character string enclosed in quotes. It does not have to be the same length as the macro variable. It can be up to 1,000 characters long.

variable — Code a macro variable declared CHARACTER. It can be up to 1,000 characters long. The variable can be subsringed. If the variable is in an array, it must be subscripted.

comparison — Code the type of comparison. You can use one of the following:

Operator	Meaning
=	equal to
¬=	not equal to
>	greater than
<	less than
>=	greater than or equal to
<=	less than or equal to
¬>	not greater than
¬<	not less than

|| — You can concatenate character constants and macro variables with other character constants and macro variables using the concatenation operator (||). This operator simply connects the operands together forming a string with a length equal to the sum of the lengths of the operands. The total length can be up to 1,000 characters.

Note: If the macro variable and the value it is compared to are not the same length, the comparison adds blanks on the right of the shorter value, to make it equal in length to the longer value.

The Null Statement in Macro Definition Code

Purpose

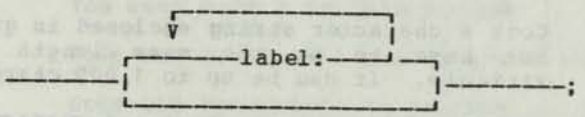
A statement that specifies a macro phase statement without any macro processing action to be performed.

Rules

When the target of a GOTO statement is a null statement, processing proceeds with the statement following the null statement.

When a IF statement has a null statement as the THEN or ELSE clause, no action occurs when the comparison in the IF statement leads to the null statement.

Syntax



Operand

label Optional. Code one or more labels, each followed by a colon. (A label is required if the null statement is the target of a GOTO or GO TO statement.)

%MACRO -- Start a Macro Definition

Purpose

You use a %MACRO statement to start a macro definition, to specify the name or names of the macro you are defining, and to specify the keynames of the arguments that you might pass to the macro when you invoke it.

Rules

You code after this statement the macro definition statements that prescribe the set of logic that belongs to this macro. You finish the definition with a %END statement.

You invoke a macro definition for execution of its statements during the macro processing phase of the compiler with an invocation statement. See, "Invoking a Macro Definition."

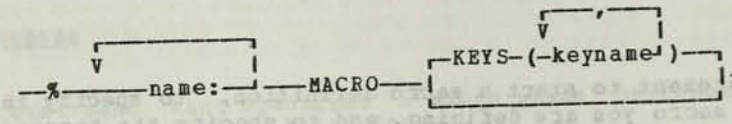
See the MACPARM macro definition function and MACPARM compiler control statement for information that can be passed by the invoker for use in the macro definition.

Besides keyname parameters, a macro invocation can pass one or more variables called positional parameters for use by the invoked macro. These positional parameters are not defined in the MACRO statement.

Because the macro processing phase of the compiler uses only one pass, each uncataloged macro definition must appear earlier in your source text than your invocation of that macro.

During the macro processing phase of the compiler, MACRO Is a reserved keyword; do not use it as a variable name.

Syntax



Operands

name — Code one or up to 16 names that can be used to invoke this macro. The maximum size allowed for a name is eight characters. The first character of each name must be alphabetic (A to Z) and the remaining characters are alphanumerics (A through Z, 0 through 9).

See the MACNAME macro definition function for use with a multiple-name macro definition.

keyname — Code the name or names of parameters that the invoking statement can pass to the macro.

For the duration of a macro definition's execution, each keyname specified in this macro definition's %MACRO statement has a temporary macro definition function of returning what this macro's invocation specified for this keyname. See, "Keyname -- Macro Definition Function for Invocation Arguments." You can defeat this function by coding the keyname in macro text within single quotes. This makes it a literal that is not scanned for special meaning. This is illustrated in an example in the ANSWER statement description.

Examples of the MACRO statement:

```
%TEST1:TEST2:TEST3:  MACRO;
   /* THIS MACRO DEFINITION CAN BE INVOKED USING ANY OF THESE NAMES */

%LOAD:MACRO KEYS(EP,EPLOC,DE,DCB);
   /* INDICATES WHAT KEYNAME PARAMETERS CAN APPEAR IN THE
        INVOKING STATEMENT */
```

For the above example, the macro invocation might look like this:

```
     ?LOAD  EP(BEGIN)  DCB(DCBADDR);
```

RETURN -- Stop Macro Execution and Return to Invoker

Purpose

You use the macro definition RETURN statement to stop processing within the macro, and you can optionally pass a severity code to indicate the seriousness of the condition that the macro has detected during this execution.

Rules

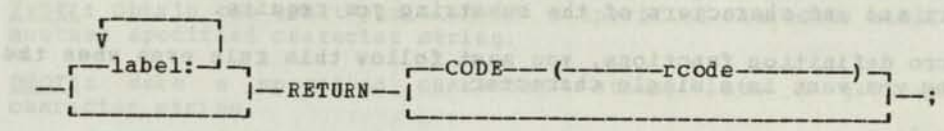
The RETURN statement causes the macro processing phase to immediately leave the macro definition it is executing, and to resume scanning text after the semicolon that ends the invocation of the macro that issued RETURN.

If the macro invocation was from macro outer code, the RETURN statement causes scanning to resume at the next macro outer statement.

If a RETURN statement with a severity code of 16 or more is executed, the compiler stops after the macro processing phase.

RETURN is a reserved keyword; do not use it as a variable name.

Syntax



Operand

label — Optional. Code one or more labels, each followed by a colon.

CODE — You code this keyword to indicate that you are passing a severity code with the return statement.

rcode — Code an arithmetic expression. Note the effect of this value under "Rules", above.

Note: The rcode value is not accessible by the macro code. To make the value of the return code value available for examination by the macro logic that follows the RETURN action, declare an external variable and assign to it the value of rcode just before issuing RETURN.

Inform the users of the macro of the significance of the return values available in this variable declared external in their code.

USING MACRO DEFINITION BUILT-IN FUNCTIONS

The term "macro definition functions" refers to built-in functions that are available for macro definition code only. When a macro with macro definition statements that contain macro definition functions is invoked during macro processing, the macro processing phase of the compiler replaces the macro definition functions with the values they stand for during its execution of the macro.

There are three general classes of macro definition functions:

- Invocation data functions
- Compiler and machine constant functions
- String handling functions.

Some of the macro definition functions return fixed values, and some return character values. See the FIXED and CHAR functions for converting one to the other, where appropriate.

You can code the macro definition functions anywhere you can use a variable, except as the receiver of an assignment.

If you declare a macro variable with the same name as a macro definition function, you cannot use that function.

SUBSTRINGING SINGLE CHARACTERS IN MACRO DEFINITION FUNCTIONS

When you code a substring function in PL/DS, you specify the position of the start and end characters of the substring you require.

With macro definition functions, you must follow this rule even when the substring you want is a single character.

For example,

```
MACNAME(2:2)
```

...returns the second character of the MACNAME function's substring.

SUMMARY OF MACRO DEFINITION FUNCTIONS

Keyname: Obtain the argument in this macro's invocation for one of the keyname parameters of this macro definition.

MACCOL: Obtain the macro invocation column number where this macro's name started.

MACINDEX: Obtain the number of macro invocations that have occurred so far in this macro processing phase.

MACKEYS: Obtain from this macro's invocation the keynames and their specified values.

MACLABEL: Obtain the labels on this macro's invocation.

MACLIST: Obtain from this macro's invocation the positional (non-keyword) arguments.

MACNAME: Obtain from this macro's invocation the name by which this macro was invoked.

NUMBER: Obtain from this macro's invocation the number of arguments that were submitted.

MACLMAR: Obtain the left source margin for this compiler run.

MACRMAR: Obtain the right source margin for this compiler run.

COMMENT: Make a comment on the listing from the specified text.

INDEX: Obtain the start position of a specified character string within another specified character string.

QUOTE: Make a specified character string into a quotes-enclosed character string.

REPEAT: Reiterates a specified character string and then repeats it a specified number of times.

CHAR: Converts specified fixed value to a character string.

FIXED: Converts specified character value to a fixed value.

MACTIME: Obtains the time of this compiler run.

MACDATE: Obtains the date of this compiler run.

MACPARM: Obtains the character string passed to this macro processing phase in the MACPARM parameter of the EXEC card's PARM field.

INVOCATION DATA MACRO DEFINITION FUNCTIONS

The invocation data definition functions provide information, during macro definition execution, from the macro invocation of the macro definition you code them in. These functions are described in alphabetic sequence on the following pages, with some examples.

Note: For the duration of a macro definition's execution, each of the invocation functions shown in the figure below has as a temporary value the corresponding value from this macro's invocation only. (See the descriptions of the individual functions for and defaults, where applicable.)

Here is a diagram of the parts of a macro invocation, an example of a macro invocation, and the macro definition function(s) applicable to each part.

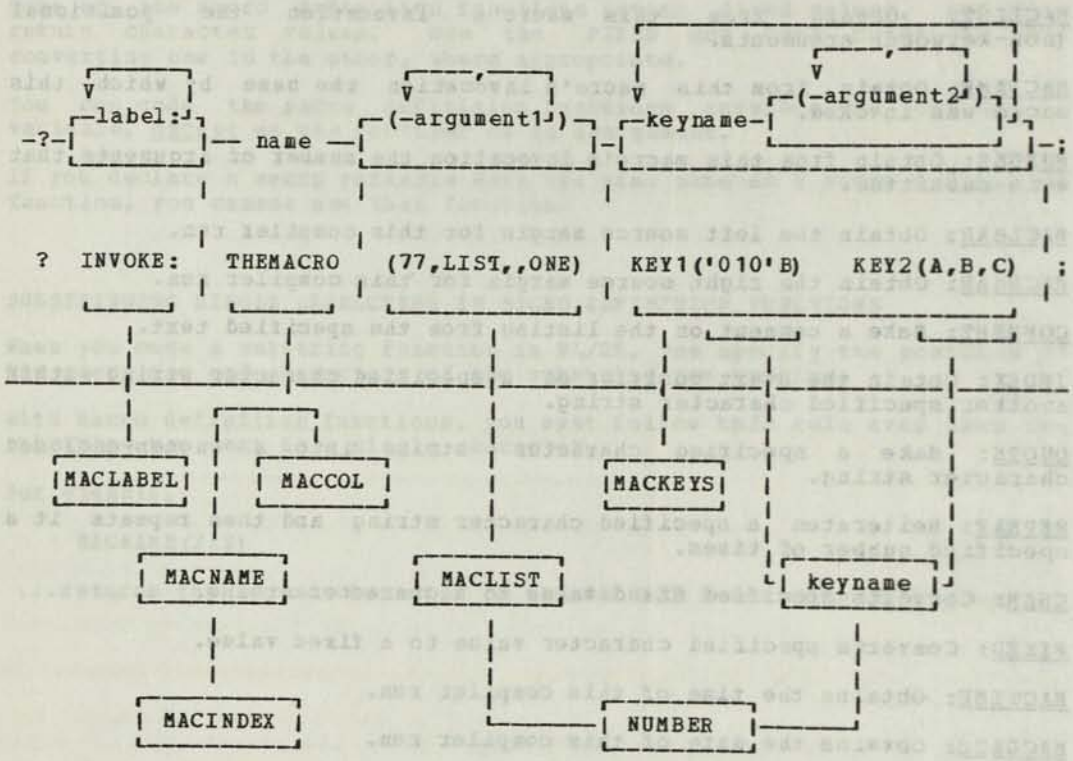


Figure 24. Invocation Data Returned By Macro Invocation Functions

Macro invocations are described in this publication under "Invoking Macro Definitions."

Keyname -- Macro Definition Function for Invocation Arguments

Purpose

You use the keyname function in a macro definition to obtain one or more of the arguments passed by the macro invocation for one of this macro's keyword parameters. (The keyword parameters originate in the %MACRO statement.) See Figure 24.

Rules

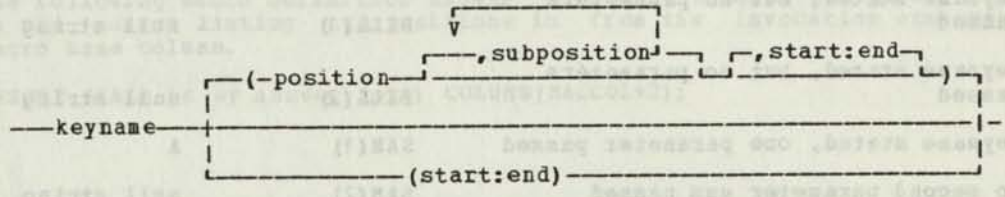
The data type of the result of the keyname function is a character string.

The string that is returned includes the parentheses coded around the argument in the invocation.

For a keyname that the macro invocation stated, but passed no characters for, the function gives the keyname.

For a keyname that the macro invocation did not even state, the function gives a null string.

Syntax



Operands

keyname Code the keyname whose invocation parameter(s) you want.

Use one of the parameter list keynames specified for this macro in its %MACRO statement.

If you code no more operands, this statement gives the value specified for this keyname in the macro's invocation.

position Optional. You can specify that you want one of the parameters from the list of parameters that the macro invocation specified for this keyname. The keyname returned is enclosed in one outer set of parentheses and separated by commas. Code a fixed variable or constant or a complex arithmetic expression.

subposition Optional. For the parameter in position, above, specify the subparameter, or sub-subparameter, etc., that you want. Code a fixed variable, a constant, or a complex arithmetic expression.

start:end Optional. For keyname substrings.

1. With the position and optional subposition operands, code the position numbers of the start and end characters of the substring you want.

2. By itself right after keyname, specify the start and end positions of the substring within the entire invocation parameter list submitted for this keyname.

For a single position, code its number in both start and end.

Example

Given this macro definition statement:

```
%TRY: MACRO KEYS(BILL, JOE, HARRY, PAUL, SAM)
    - - -
    - - -
    - - -
%END;
```

and this macro invocation:

```
? TRY BILL SAM(A) JOE (A,B,(C)) PAUL()
```

...The following are some possible keyname macro definition functions and the results that the compiler returns:

DESCRIPTION	KEYNAME FUNCTION	RESULT
Keyname stated, but no parameters passed	BILL	BILL
Keyname stated, but no parameters passed	BILL(1)	null string
Keyname stated, but no parameters passed	BILL(2)	null string
Keyname stated, one parameter passed	SAM(1)	A
No second parameter was passed	SAM(2)	null string
When no parameter specified, default all	JOE	(A,B,(C))
First parameter in string	JOE(1)	A
Second parameter in string	JOE(2)	B
The nested parentheses are in a sublist	JOE(3)	(C)
With one parameter	JOE(3,1)	C
With one sub-sublist	JOE(3,1,1)	C
And no second sub-sublist	JOE(3,1,2)	null string
First level string specified in invocation	PAUL	()
But no sublist 1	PAUL(1)	null string
Nor sublist 2	PAUL(2)	null string
HARRY keyname not specified in macro invocation	HARRY	null string

MACCOL -- Macro Definition Function for Invocation Column

Purpose

You use the MACCOL function in a macro definition to obtain the column number where the macro's name started in this macro execution's invocation statement. You can use this value to position the answer text from this macro on the output source listing, relative to the invocation name. See Figure 24.

Rule

The data type of the result of the MACCOL function is FIXED.

If you declare MACCOL as a variable, it is no longer a macro definition function.

Syntax

```
--------MACCOL--------
```

Example

The following macro definition ANSWER statement indents the answer text on the source listing two positions in from the invocation statement macro name column.

```
ANSWER (this is my answer text) COLUMN(MACCOL+2);
```

MACINDEX -- Macro Definition Function for Invocation Total

Purpose

You use the MACINDEX function in a macro definition to obtain the number of macro invocations that have occurred in the current macro processing phase of the compiler. See Figure 24.

Rules

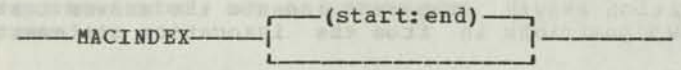
The data type of the result of the MACINDEX function is character.

The first time MACINDEX is used, the value is 0001. Each time a macro is invoked, the value of MACINDEX is incremented by 1; however, as long as a macro definition has not returned control to its invoker, this macro sees a constant value for MACINDEX.

The value is a four-digit string unless the substring parameter changes this.

If you declare MACINDEX as a variable, it loses its macro definition function.

Syntax



Operands

start:end — Optional. You may code any substring expression indicating the start position and the end position of a portion of the MACINDEX string. The start and end may be specified as FIXED variables, decimal constants, or arithmetic expressions. For a single position, code its number in both start and end.

Example

Assume the macro phase is in progress, and the value of MACINDEX is 0100. The macro definition named OUTER is invoked. The value of MACINDEX during the execution of OUTER and the execution of INNER that it invokes are shown in comments.

```
%OUTER: MACRO;
   DCL (A,C) CHARACTER;
       - - -
       - - -
       - - -
   A = MACINDEX;            /* A=0101 */
   ANSWER ('?INNER;');      /* INVOKE ANOTHER MACRO DEFINITION */
   C = MACINDEX;            /* C=0101 */
       - - -
       - - -
       - - -
%END;

%INNER: MACRO;
   DECLARE B CHARACTER;
       - - -
       - - -
       - - -
   B = MACINDEX;            /* B=0102 */
       - - -
       - - -
       - - -
%END;
```

Now assume that a further macro, %NEWOUTER, is invoked following %OUTER:

```
%NEWOUTER: MACRO;
   DCL (D) CHARACTER;
       - - -
       - - -
   D = MACINDEX;            /* D =0103 */
       - - -
       - - -
%END;
```

This demonstrates that for every macro definition, MACINDEX returns a unique value that you can use without fear of duplication throughout the macro processing phase of PL/DS.

MACKEYS -- Macro Definition Function for Invocation Keyname Arguments

Purpose

You use the MACKEYS function in a macro definition to obtain a string of all the keynames and their values that were specified in the macro invocation statement. See Figure 24.

Rules

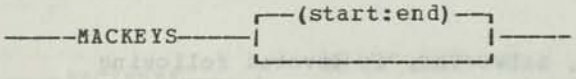
The data type of the result of the MACKEYS function is character.

Comments and non-significant blanks are removed from the list that is returned.

Do not use the NUMBER macro definition function with MACKEYS to determine the number of keys specified in the invocation.

If you declare MACKEYS as a variable, it loses its macro definition function.

Syntax



Operands

start:end — Optional. You may code any substring expression indicating the start position and end position of a portion of the MACKEYS string. The start and end may be specified as fixed variables, decimal constants, or complex arithmetic expressions. For a single position, code its number in both start and end.

Examples

For the following invocation:

```
?LAB1: A(X) KEY1 KEY2('10')KEY3
```

...the length of MACKEYS is 10 and its value is:

```
KEY1KEY2('10')KEY3
```

You may use MACKEYS with other built-in functions that address a macro invocation to create a copy of that invocation string. The following example gives the value of MACKEYS for a specific macro invocation. When this ANSWER statement is executed:

```
ANS ('?'||MACLABEL||'MACB '||MACLIST||MACKEYS||';');
```

...it produces the following call to an inner macro, MACB:

```
?LAB1:MACB (X)KEY1 KEY2('10')KEY3;
```

Note that the inner macro name is followed by a blank to allow for a null MACLIST.

In the following example, MACKEYS is used to provide a reconstructed macro invocation string as commentary in PL/DS code.

Given this macro invocation for MAC1:

```
? LAB1: MAC1(A,B) KEY1(X) KEY2 KEY3('AB');
```

...and given these two statements in the MAC1 macro code:

```
IDENT = MACLABEL||MACNAME||MACLIST||MACKEYS;

ANS('DO;'||COMMENT(IDENT));
```

...the macro processing phase generates this text:

```
DO;/*LAB1:MAC1(A,B)KEY1(X)KEY2 KEY3('AB')*/
```

MACLABEL -- Macro Definition Function for Invocation Label

Purpose

You use the MACLABEL function in a macro definition to obtain the label on the macro invocation statement for this macro, or a portion of it. See Figure 24.

Rules

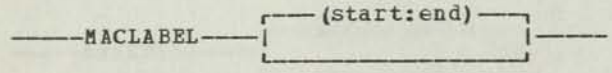
The data type of the result of the MACLABEL function is character.

Any blanks or comments that were between the labels on the macro invocation are removed; the colon after each label remains.

If the macro invocation had no label, the result is a null.

If you declare MACLABEL as a variable, it loses its macro definition function.

Syntax



Operand

start:end — Optional. You may code any substring expression indicating the start position and end position of a portion of the MACLABEL string. The start and end may be specified as fixed variables, decimal constants, or complex arithmetic expressions. For a single position, code its number in both start and end.

Example

Given the CHARACTER macro variable A and the macro invocation for MAC1:

```
DCL A CHAR;
? ABC:DEF:123: MAC1;
```

If you code the MACLABEL function within MAC1:

```
A=MACLABEL(3:9)
```

It returns

```
C:DEF:1
```

MACLIST -- Macro Definition for Invocation Positional Arguments

Purpose

You use the MACLIST function in a macro definition to find the value of one or more of the positional arguments in the macro invocation statement for this macro. See Figure 24.

Rules

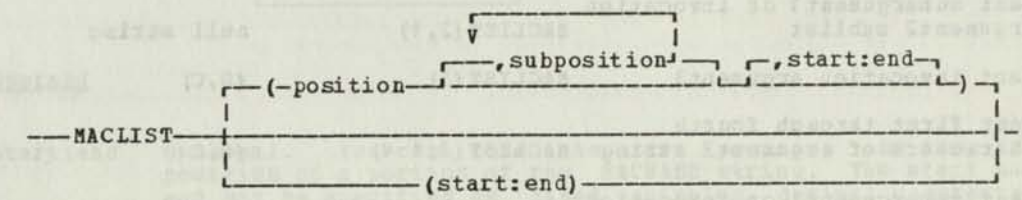
The data type of the result of the MACLIST function is character.

If no positional arguments and enclosing parentheses are passed, the result of the function is a null string.

Any blanks or comments that were between the arguments on the macro invocation are removed; the sublist parentheses and the comma after each sublist argument remain.

If you declare MACLIST as a variable, it loses its macro definition function.

Syntax



Operands

position — Optional. You can code the position of an argument or sublist within the macro definition's positional parameter list. The position can be specified as a variable, constant, or complex arithmetic expression.

subposition — Optional. You can code the position of the argument you want within the sublist specified in the position parameter. The subposition can be specified as a fixed variable, constant, or a complex arithmetic expression.

Note: If the position is followed by a subposition value of 1 and the position is not a sublist, the subposition is ignored. The same is true when a subposition is followed by a value of 1 and no more sublist values exist.

start:end — Optional. You can code a substring expression that indicates the start position and end position of a portion of the MACLIST string. The start and end may be specified as fixed variables, decimal constants, or complex arithmetic expressions. For a single position code its number in both start and end.

Examples

Given a macro invocation that passes the following positional parameter list:

```
(A,,(B,C),(D))
```

...the following lists the values returned by various uses of the MACLIST Built-in function in the invoked macro.

DESCRIPTION	MACLIST FUNCTION	RESULT
Want all invocation arguments	MACLIST	(A,,(B,C),(D))
Want second through fourth characters of positional argument string	MACLIST(2:4)	A,,
Want invocation argument1	MACLIST(1)	A
Want subargument1 of invocation argument1 sublist	MACLIST(1,1)	A
Want subargument2 of invocation argument1 sublist	MACLIST(1,2)	null string
Want invocation argument2 (second comma delimits it)	MACLIST(2)	null string
Want subargument1 of invocation argument2 sublist	MACLIST(2,1)	null string
Want invocation argument3	MACLIST(3)	(B,C)
Want first through fourth characters of argument3 string	MACLIST(3,1:4)	(B,C
Want subargument1 of invocation argument3 sublist	MACLIST(3,1)	B
Want subargument2 of invocation argument3 sublist	MACLIST(3,2)	C
Want subargument3 of invocation argument3 sublist	MACLIST(3,3)	null string
Want invocation argument4	MACLIST(4)	(D)
Want subargument1 of invocation argument4 sublist	MACLIST(4,1)	D
Want subsubargument1 of subargument1 of invocation argument4 sublist	MACLIST(4,1,1)	D

MACNAME -- Macro Definition Function for Invocation Name

Purpose

You use the MACNAME function in a macro definition to obtain the name, out of the list of names in the macro definition, that the macro invocation used to invoke this macro. You can use multiple macro names to modify the processing within the macro depending on the name that was used to invoke it. See Figure 24.

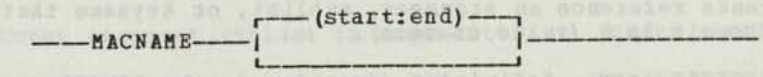
Rules

The data type of the result of the MACNAME function is character.

The length of the result is the number of characters in the name or in the substring, if specified.

If you declare MACNAME as a variable, it loses its macro definition function.

Syntax



Operand

start:end — Optional. You can code the start position and the end position of a portion of the MACNAME string. The start and end may be specified as fixed variables, decimal constants, or complex arithmetic expressions. For a single position, code its number in both start and end.

Example

When this macro:

```
%TEST1:   TEST2: MACRO;
DCL A CHAR;
A=MACNAME;
    - - -
    - - -
    - - -
%END;
```

...is invoked during the macro phase with:

```
?TEST2;
```

...the variable A is assigned the value 'TEST2' during execution.

NUMBER -- Macro Definition Function for Invocation Argument Quantity

Purpose

You code the NUMBER function in a macro definition to obtain one of:

- the number of positional arguments in the invocation.
- the number of arguments with a specified keyname in the invocation.
- the number of arguments in a parameter list sublist.

See Figure 24.

Rules

The data type of the result of the NUMBER function is fixed.

When the NUMBER operands reference anything that is enclosed in parentheses, the result is 1 plus the number of commas within the parentheses; commas included in sublists within the reference are not added into the value.

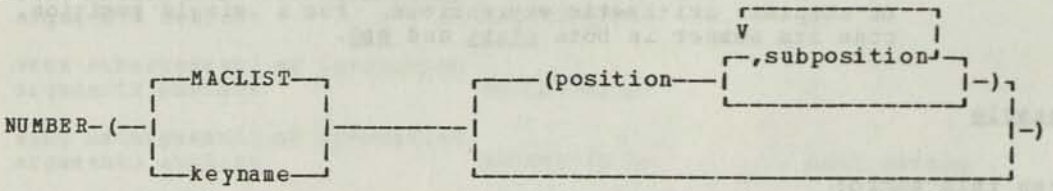
When the NUMBER operands reference an argument, keyname, or sublist that is not within parentheses, the result is 1.

When the NUMBER operands reference an argument, sublist, or keyname that does not exist, the result is 0 (value of zero).

Do not use the MACKEYS macro definition function with NUMBER to determine the number of keys submitted in the invocation.

If you declare NUMBER as a variable, it loses its macro definition function.

Syntax



Operands

MACLIST Code this keyword if the arguments that you want counted are in a positional macro list.

keyname If the arguments that you want counted are in a keyname parameter list, code the name of the key.

position Optional. For the MACLIST keyword, code the position of the argument that is a sublist you want counted.

For the keyname parameter, code the position of the argument in that keyname's list that is a sublist you want counted. The position parameter can be a fixed variable, a constant, or a complex arithmetic expression.

subposition
Optional. For the parameter in position, above, specify the subparameter, sub-subparameter, etc., that you want. Code a fixed variable, a constant, or a complex arithmetic expression.

Example

When the macro definition TEST is invoked with the following macro invocation statement:

```
?TEST    (A,,(B,C),(D))  JOE(A,B)  SAM(A)  BILL  HAL(A,(B,(C,D)));
```

...there are four positional parameters and four keyword parameters.

The following is a list of the values returned by various uses of the NUMBER function within the macro TEST:

DESCRIPTION	NUMBER FUNCTION	RESULT
Count positional arguments	NUMBER(MACLIST)	4
Count argument sublist in positional argument1	NUMBER(MACLIST(1))	1
Count argument sublist in positional argument2	NUMBER(MACLIST(2))	0
Count argument sublist in positional argument3	NUMBER(MACLIST(3))	2
Count arguments in subposition 1 of positional argument3	NUMBER(MACLIST(3,1))	1
Count argument sublist in argument4	NUMBER(MACLIST(4))	1
Count arguments for keyword JOE	NUMBER(JOE)	2
Count arguments in JOE's argument1 sublist	NUMBER(JOE(1))	1
Count arguments in JOE's argument3 sublist	NUMBER(JOE(3))	0
Count arguments for keyword SAM	NUMBER(SAM)	1
Count arguments in SAM's argument1 sublist	NUMBER(SAM(1))	1
Count arguments for keyword BILL	NUMBER(BILL)	0
Count arguments in BILL's argument1 sublist	NUMBER(BILL(1))	0
Count arguments in BILL's argument2 sublist	NUMBER(BILL(2))	0
Count arguments in HAL's argument1 sublist	NUMBER(HAL(1))	1
Count arguments in subposition1 of HAL's argument1 sublist	NUMBER(HAL(1,1))	1
Count arguments in HAL's argument2 sublist	NUMBER(HAL(2))	2
Count arguments in subposition1 of HAL's argument2 sublist	NUMBER(HAL(2,1))	1
Count arguments in subposition2 of HAL's argument2 sublist	NUMBER(HAL(2,2))	2

COMPILE-TIME MACRO DEFINITION FUNCTIONS

The compile-time functions obtain compiler control and computer time/date values. They are:

- MACLMAR -- Left compiler margin.
- MACPARM -- EXEC card MACPARM parameter string.
- MACRMAR -- Right compiler margin.
- MACTIME -- Computer clock time.
- MACDATE -- Computer clock date.

Descriptions and examples of each of these are on the following pages.

MACDATE -- Macro Definition Function for Compile Date

Purpose

You use the MACDATE function in a macro definition to obtain the date of the compile in which this macro is invoked.

Rules

The data type of the result of the MACDATE function is character.

The length of the result in five characters, unless you code the substring operand.

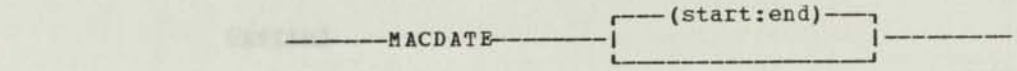
The format of the character string is yyddd, where:

yy is the year: from 00 to 99
ddd is the day-of-year date: from 001 to 366

The date value is obtained from the date set into the computer at IPL time, and updated by the computer's control program.

If you declare MACDATE as a variable, it loses its macro definition function.

Syntax



Operand

start:end — Optional. You can specify a part of the MACDATE string. Code any substring expression indicating the start position and end position of the MACDATE string you want. The start and end may be fixed variables, decimal constants, or complex arithmetic expressions. For a single position, code its number in both start and end.

MACLMAR -- Macro Definition Function for Left Source Margin

Purpose

You use the MACLMAR function in a macro definition to obtain the left source margin that is in effect for this compiler run. The left source margin is the column number in the source code records where the compiler begins to read. This allows you to adjust the left source margin of your answer text.

Rules

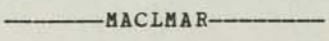
The data type of the result of the MACLMAR function is fixed.

The default left source margin is 2. This can be changed with the MARGINS compiler option.

See the MACRMAR function for the right source margin.

If you declare MACLMAR as a variable, it loses its macro definition function.

Syntax



Example

In this example, FIXED macro variable A is assigned the value of the left source margin and used to make the answer text begin at the proper source margin.

```
DECLARE A FIXED;
  - - -
  - - -
  - - -
A = MACLMAR;
  - - -
  - - -
  - - -
ANSWER(expression) COLUMN(A);  /* THE ANSWER TEXT WILL APPEAR
          IN THE SOURCE TEXT BEGINNING IN THE LEFT SOURCE MARGIN
          USED FOR THIS COMPILATION  */
```

MACPARM -- Macro Definition Function MACPARM Compiler Option String

Purpose

You use the MACPARM function in a macro definition to obtain the character string, or optionally part of it, submitted as the parameter of the MACPARM compiler option in the compiler run in which this macro is invoked.

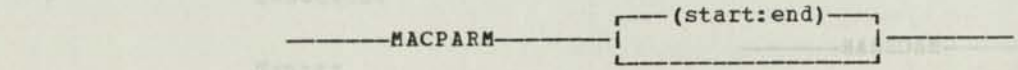
Rules

The data type of the result of the MACPARM function is character.

The MACPARM compiler option is for passing a character string for reference by the MACPARM macro definition function. You assign the meanings or controls implied by certain character strings, and implement them in your macro.

If you declare MACPARM as a variable, it loses its macro definition function.

Syntax



Operand

start:end — Optional. You can specify a part of the MACPARM string. Code any substring expression indicating the start position and end position of the portion of the MACPARM string that you want. The start and end may be fixed variables, decimal constants, or complex arithmetic expressions. For a single character, code its number in both start and end.

Example

In this example, an EXEC card specifies a MACPARM value of ABCDE, and in a macro the MACPARM function is used to establish a "go to finish" decision.

```
//STEP EXEC PGM=IRE00000,PARM='MACPARM(''ABCDE'')'
          - - -
          - - -
          - - -
%JOB:MACRO;
   DCL (P) CHAR;
   P = MACPARM;                  /* P = ABCDE    */
   IF P¬='ABCDE' THEN            /* TEST MACPARM*/
       GOTO FINISH;
       - - -
       - - -
FINISH:;
%END;
```

MACRMAR -- Macro Definition Function for Right Source Margin

Purpose

You use the MACRMAR function in a macro definition to obtain the right source margin that is in effect for this compiler run. The right source margin is the column number in the source code records where the compiler stops reading. This allows you to adjust the position of answer text with respect to the right source margin.

Rules

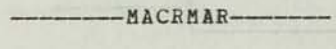
The data type of the result of the MACRMAR function is fixed.

The default right source margin is 72. This can be changed with the MARGINS compiler option.

See the MACLMAR function for the left source margin.

If you declare MACLMAR as a variable, it loses its macro definition function.

Syntax



Example

In this example, answer text (a microfiche flag, in this case) is positioned at the right margin.

```
DECLARE A FIXED;
DECLARE FLAG CHARACTER;
    - - -
    - - -
    - - -
A = MACRMAR - LENGTH(FLAG) - 1;
    - - -
    - - -
    - - -
ANSWER(FLAG) COLUMN(A); /* OUTPUT FLAG NEXT TO RIGHT MARGIN */
```

MACTIME -- Macro Definition Function for Compile Time

Purpose

You use the MACTIME function in a macro definition to obtain the time of day of the compile in which this macro is invoked.

Rules

The data type of the result of the MACTIME function is character.

The length of the result is six characters, unless you code the substring operand.

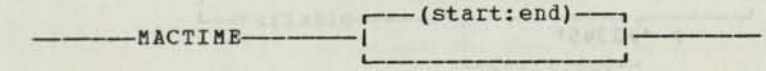
The format of the character string is hhmmss, where:

hh is the hour: from 00 to 23
mm is the minute: from 00 to 59
ss is the second: from 00 to 59

The time value is obtained from the time set into the computer at IPL time, and updated by the computer's control program.

If you declare MACTIME as a variable, it loses its macro definition function.

Syntax



Operand

start:end — Optional. You can specify a part of the MACTIME string. Code any substring expression indicating the start position and end position of the portion of the MACTIME string that you want. The start and end may be fixed, variables, decimal constants, or complex arithmetic expressions. For a single position, code its number in both start and end.

STRING HANDLING MACRO DEFINITION FUNCTIONS

The character string handling macro definition functions perform character-string-related operations. There are seven functions:

- COMMENT makes a comment out of a character string.
- INDEX obtains the position of a character string in another character string.
- LENGTH obtains the length of a string.
- QUOTE encloses a character string in quotes.
- REPEAT repeats a character string a specified number of times.
- CHAR converts fixed data to character data.
- FIXED converts character data to fixed data.

Descriptions and examples of each are on the following pages.

Here is a summary of the results of each function. Given the character string '+2345' assigned to variable X,

```
X='+2345'
```

the result of each function is:

```
COMMENT(X)    --> /*+2345*/

INDEX(X,'45')--> 4

LENGTH(X)     --> 5

QUOTE(X)      --> '+2345'

REPEAT(X,1)   --> +2345+2345

CHAR(X)       --> +2345 (character)

FIXED(X)      --> 2345 (arithmetic)
```

COMMENT -- Macro Definition Function to Generate a Comment

Purpose

You use the COMMENT function in a macro definition to issue a character string as a comment in answer text. (You can code your own comments without the COMMENT function, but COMMENT ensures that there are no stray /* or */ within the comment text.)

Rules

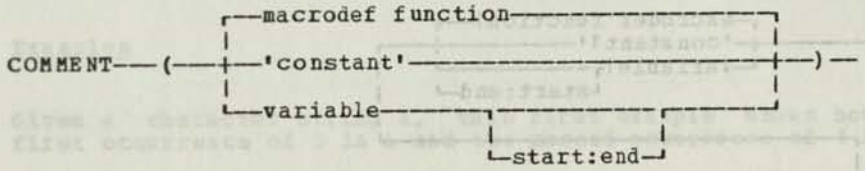
The data type of the result of the COMMENT function is character.

If any comment delimiters, /* or */, appear within the specified character string, the COMMENT function changes them to /> or </, respectively.

The resultant character string must be no more than 1,000 characters.

If you declare COMMENT as a variable, it loses its macro definition function.

Syntax



Operands

macrodef function	You may code a macro definition function if the value provided by the function is a character string.
'constant'	Constant. May be any character string enclosed in single quotes. The maximum length is 996 characters.
variable	Variable should be a CHARACTER variable that has previously been assigned a character string.

Example

In this example, two macro variables, A and B, are declared CHARACTER, and assigned character strings that are comments.

```
DCL A CHARACTER;
DCL B CHARACTER;
A = COMMENT('WHAT');
B = COMMENT('NAME /*ODD''QUOTE*/');
```

...The value of A is /*WHAT*/ and the value of B is /*NAME/>ODD'QUOTE</*/.

INDEX -- Macro Definition Function for Character String Occurrences

Purpose

You use the INDEX function in a macro definition to obtain if and where a specified character string or, optionally, the nth occurrence of a recurring character string, is located in another specified searched string.

Rules

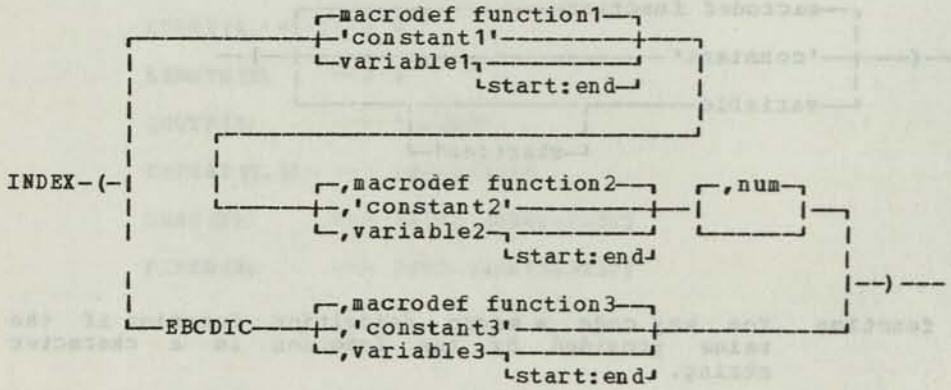
The data type of the result of the INDEX function is fixed.

The result is the number of positions from the start of the searched string of the starting position of the specified searched-for string.

If there is no occurrence of the searched-for string, the result is a zero.

If you declare INDEX as a variable, it loses its macro definition function.

Syntax



Operands

macrodef function1 This is the first of three ways you can specify the character string to be searched. Code a macro definition function with a character string result.

'constant1' This is the second of three ways you can specify the character string to be searched. Code the character string, enclosed in single quotes.

variable1 This is the third of three ways you can specify the character string to be searched. Code a macro variable declared CHARACTER that has been assigned the character string.

macrodef function2 This is the first of three ways you can specify the character string to be searched for. Code a macro definition function with a character string result.

'constant2' This is the second of three ways you can specify the character string to be searched for. Code the character string, enclosed in single quotes.

variable2 — This is the third of three ways you can specify the character string to be searched for. Code a macro variable declared CHARACTER that has been assigned the character string.

num — Optional. You can specify which occurrence of a recurring searched-for string2 you want the starting position of in the searched string1. It may be specified as a fixed variable, decimal constant, or complex arithmetic expression. If you omit the num operand, its default value is 1.

EBCDIC — You code the keyword EBCDIC to specify that you want the position (decimal number) of a character in the EBCDIC table.

macrodef function3 — This is the first of three ways to specify the character you are searching the EBCDIC table for. The macro definition function you code must have a single character result.

'constant3' — This is the second of three ways to specify the character you are searching the EBCDIC table for. Code the single character, enclosed in single quotes.

variable3 — This is the third of three ways to specify the character you are searching the EBCDIC table for. Code a macro variable declared CHARACTER that has been assigned the single character.

Examples

Given a character string A, this first example shows how to find the first occurrence of 5 in A and the second occurrence of 1 in A.

```
DCL A CHAR;
DCL B FIXED;
DCL C FIXED;
A = '0123456789';
B = INDEX(A,'5');
C = INDEX(A,'1',2);
```

The value of B is 6 and the value of C is 0 because there is no second occurrence of the character 1.

This second example illustrates the conversion of one character to an arithmetic value.

This form of the INDEX macro definition function converts one character into an arithmetic value based on the EBCDIC internal representation of the character.

```
DCL D FIXED;
D = INDEX(EBCDIC,'$');
```

The value of D is 91, which is the arithmetic value for the character $.

LENGTH -- Macro Definition Function for Character String Length

Purpose

You use the LENGTH function in a macro definition to obtain the length of the character string you specify.

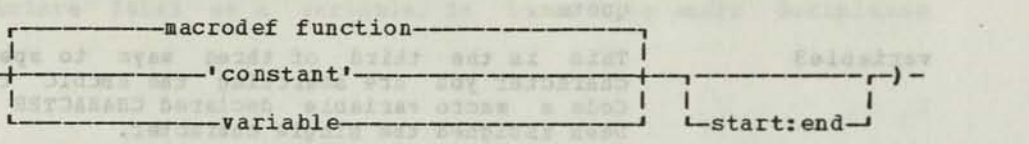
Rules

The data type of the result of the LENGTH function is fixed.

If the specified string is a null string, the result is 0 (zero).

If you declare LENGTH as a variable, it loses its macro definition function.

Syntax



Operands

macdef function	You may specify a macro definition function that returns a character string.
'constant'	You may code any character string, surrounded by single quotes. The string can be up to 1,000 characters.
variable	You may code a macro variable declared CHARACTER that you have assigned a character string to.

Examples

```
LENGTH('ABC') is 3.
LENGTH('AB''C')) is 4.

DCL A CHAR;
A='HLB';
LENGTH(A) is 3.

?MACRO1 (A,,(B,C),(D))   JOE(A,B);
LENGTH(MACLIST) is 14.
LENGTH(MACLIST(1)) is 1.
LENGTH(MACLIST(3)) is 5.
LENGTH(MACLIST(3,1)) is 1.
LENGTH(JOE) is 5.
LENGTH(COMMENT(MACLIST(3))) is 9. /*(B,C)*/
                                  |
                                  └->123456789
```

QUOTE -- Macro Definition Function to Build a Literal

Purpose

You use the QUOTE function in a macro definition to enclose a specified character string in single quotes. (You can code your own single-quote-enclosed character string, but QUOTE ensures that all single quotes in the string are doubled.)

Rules

The data type of the result of the QUOTE function is character.

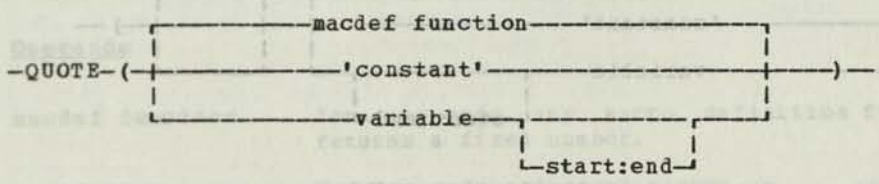
The contents of the result is a character string, with single quotes as the first and last characters.

Each single quote in the specified character string becomes two single quotes. This follows the rule for quoted (or literal) quotes.

The result can be up to 1,000 characters long.

If you declare QUOTE as a variable, it loses its macro definition function.

Syntax



Operands

macdef function	You may code a macro definition function that returns the character string.
'constant'	You may code the character string of up to 998 characters, enclosed by single quotes.
variable	You may code a macro variable declared CHARACTER that you have assigned the character string to.

Example

```
DCL A CHAR;
DCL B CHAR;
DCL C FIXED;
DCL D CHAR;
DCL E CHAR;
A = 'IT''S';                    /* IT'S IS ASSIGNED TO A          */
B = QUOTE(A);                   /* 'IT''S' IS ASSIGNED TO B       */
C = 10;
D = QUOTE(CHAR(C));             /* '10' IS ASSIGNED TO D          */
E = QUOTE(QUOTE('ABC'));        /* '''ABC''' IS ASSIGNED TO E     */
ANSWER ('Y=E;');                /* E IS REPLACED WITH '''ABC''' */
                                /* ANSWER TEXT GENERATED IS:
                                                  'Y='''ABC''';'*/
```

REPEAT -- Macro Definition Function to Repeat a String

Purpose

You use the REPEAT function in a macro definition to build a character string that consists of a given character string repeated a specified number of times.

Rules

The data type of the result of the REPEAT function is character.

The content of the result is the specified character string followed by its repetition the specified number of times.

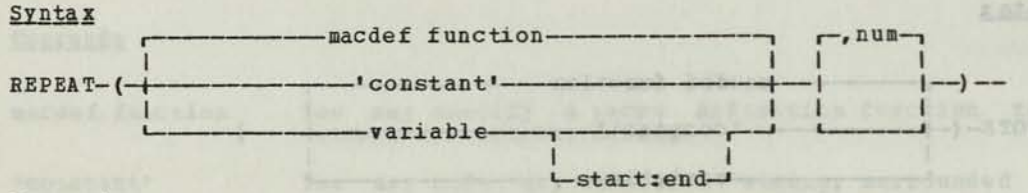
If the specified repetition number is 0 (zero), the string occurs once in the result.

The specified repetition number must not be negative.

The resultant string can be up to 1,000 characters long.

If you declare REPEAT as a variable, it loses its macro definition function.

Syntax



Operands

macdef function	You may specify, for repetition, a macro definition function that returns a character string.
'constant'	You may specify the actual character string you want repeated, enclosed by single quotes.
variable	You may code a macro variable declared CHARACTER, to which you have assigned the string you are repeating.
num	Optional. You code for num the number of times the character string is to be repeated. Remember, for the string to appear x times in the result, the num value must be x-1. Use a macro variable declared FIXED, a constant, or a complex arithmetic expression. If you omit num, the default is 1 (character string appears twice in the result).

Example:

In the statements below, the comments show the results of the REPEATs:

```
DECLARE(A,B,C,D,E) CHARACTER;
A = REPEAT('123',2);              /* A = 123123123 */
B = 'ABC';
C = REPEAT(B,1);                  /* C = ABCABC    */
D = REPEAT(B,0);                  /* D = ABC       */
```

CHAR -- Macro Definition Function to Convert Fixed to Character

Purpose

You use the CHAR function in a macro definition to convert a fixed value into a character string, with optional truncation or high order zeros.

Rules

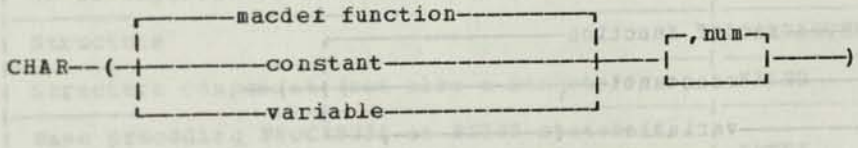
The data type of the result of the CHAR function is character.

The absolute value of the fixed data is converted.

With the num operand specified, truncation of significant digits or extension with zeros, to agree with the num value, is done on the left.

If you declare CHAR as a variable, it loses its macro definition function.

Syntax



Operands

macdef function	You may code any macro definition function that returns a fixed number.
constant	You may code a decimal number.
variable	You may specify a macro variable declared FIXED that has a value assigned to it.
num	Optional. You may specify the number of digits in the resulting character string. It can be a fixed variable, decimal constant, or complex arithmetic expression. If you omit the num operand, the default is either: the number of significant digits, if a fixed constant is converted; or the number of significant digits assigned, if a fixed variable is converted.

Example

In the following sequence, the results of various uses of CHAR are given in comments.

```
DCL V FIXED;
DCL W FIXED;
DCL X CHARACTER;
DCL Y CHARACTER;
DCL Z CHARACTER;
V = 27;
W = 123;
X = CHAR(V,4);              /* X = 0027   */
Y = CHAR(V,1);              /* Y = 7      */
Z = 'ABC'||CHAR(W);         /* Z = ABC123 */
```

FIXED -- Macro Definition Function to Convert Character to Fixed

Purpose

You use the FIXED function in a macro definition to convert a digit character string, optionally signed, into a fixed value.

Rules

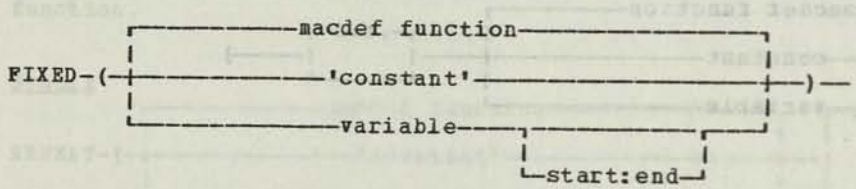
The data type of the result of the FIXED function is fixed.

To be convertible, the submitted string must be decimal digit characters; the first character may be a plus sign (+) or a minus sign (-).

The length of the fixed value is the number of digits submitted.

If you declare FIXED as a variable, it loses its macro definition function.

Syntax



Operands

macdef function	You may code any macro definition function that returns a convertible character string.
'constant'	You may code a convertible string, enclosed in single quotes.
variable	You may specify a macro variable declared CHARACTER that has a convertible value assigned to it.

Example

In the following sequence, the results of various uses of FIXED are given in comments.

```
DECLARE A CHARACTER;
DECLARE B FIXED;
DECLARE C FIXED;
DECLARE D FIXED;
DECLARE E FIXED;
A = '15';
B = FIXED(A);                /* B = 15                                  */
C = FIXED('6') + 3;          /* C = 9                                   */
D = FIXED(MACLIST(2));       /* D EQUALS THE CONVERTED VALUE OF THE
                                SECOND ARGUMENT IN THE MACRO LIST       */
E = FIXED(A(2:2));           /* E=5                                     */
```

APPENDIX A. DEFAULT AND INCOMPATIBLE DATA ATTRIBUTES

This appendix lists the default for each type of data attribute, and shows those attributes that are incompatible within the same declaration.

DEFAULT DECLARE STATEMENT VARIABLE ATTRIBUTES

The following tables show the default values that are assumed when you omit one or more data attributes on the DECLARE statement. Defaults are applied at the first appearance of the variable name.

DEFAULT DATA TYPES

Known Characteristic	Default Data Type
Scalar or Array	FIXED
Name precedes "->" or is argument of BASED attribute	POINTER
Structure	CHARACTER
Structure component (not also a structure)	FIXED
Name preceding PROCEDURE or ENTRY statement, or target on a CALL statement.	ENTRY
Name preceding any statement but PROCEDURE ENTRY statement, or GOTO statement target.	LABEL
Name declared with OPTIONS attribute	ENTRY

Figure 25. Table of DECLARE Default Data Types

DEFAULT PRECISION OR LENGTH

Known Characteristic	Default Precision or Length
FIXED	Precision is 15
POINTER	Precision is 32
BINARY	Precision is 15
CHARACTER, structure	Length is the sum of the lengths of all the components at the next logical level
BIT	Length is 1
CHARACTER	Length is 1

Figure 26. Table of DECLARE Default Precision or Length

DEFAULT SCOPE

Known Characteristic	Default Scope
Target of a CALL statement	EXTERNAL
Entry name (Except the entry name of an internal procedure)	EXTERNAL
Entry name of an internal procedure	INTERNAL
All others	INTERNAL

Figure 27. Table of DECLARE Default Scope

DEFAULT STORAGE CLASS

Declared or Implied Attributes	Default Storage Class
LOCAL	STATIC
NONLOCAL	STATIC
STATIC, if EXTERNAL and not initialized	NONLOCAL
STATIC, if INTERNAL or initialized	LOCAL
No storage class attributes specified, but EXTERNAL and not initialized	STATIC NONLOCAL
No storage class attributes specified, but INTERNAL and not initialized in a reentrant environment	AUTOMATIC
No storage class and scope attributes specified and not initialized in a reentrant environment	AUTOMATIC
No storage class and scope attribute specified and initialized in a reentrant environment	STATIC LOCAL
No storage class attributes specified, but initialized, or INTERNAL in a nonreentrant environment	STATIC LOCAL
No storage class and scope attributes specified and not initialized in a nonreentrant environment	STATIC LOCAL
Components of a structure assume the same storage class attributes as the structure. Parameters are in the parameter storage class, for which no keyword is defined.	

Figure 28. Table of DECLARE Default Storage Classes

DEFAULT BOUNDARY ALIGNMENT

Data Type	Default Boundary Alignment
FIXED (8) BINARY (8)	byte
FIXED (15) BINARY (15)	halfword
FIXED (16) BINARY (16)	halfword
FIXED (32)	word
POINTER	word
LABEL	halfword (cannot be specified)
ENTRY	halfword (cannot be specified)
CHARACTER	byte
BIT	byte (not in a structure) byte (array in a structure) bit (nonarray in a structure)

Figure 29. Table of DECLARE Default Boundary Alignment

DEFAULT POSITION

If no POSITION is specified, the default is one.

DEFAULT STRUCTURE BOUNDARY

If you do not declare a boundary for a structure, and you declare a data type for the structure, the structure has the default boundary for that data type.

If you do not declare a data type, the structure has a default boundary equal to the highest boundary required by any of its components.

DEFAULT INITIALIZATION

No initialization takes place by default.

DEFAULT RESTRICTEDNESS

Registers are RESTRICTED by default.

DEFAULT NORMALITY

All data is NORMAL by default.

INCOMPATIBLE ATTRIBUTES FOR SIMPLE ITEMS, ARRAYS, AND STRUCTURES

Certain attributes cannot be used in combination for simple items, arrays, and structures. Figures 30, 31, and 32 show all valid combinations of attributes for simple items, arrays, and structures, respectively.

COMPLEX FIGURE SYNTAX RULES

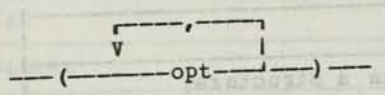
The figures of incompatible attributes follow the syntax described here.

A combination is allowed if all of the attributes lie on any single path, starting at DCL name and ending at the semicolon. The path must go through the diagram progressing from left to right and top to bottom.

Right to Left Path Restrictions

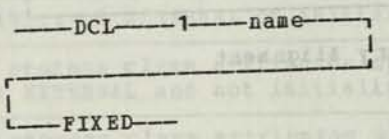
A right to left path may be taken in only two cases:

(1)



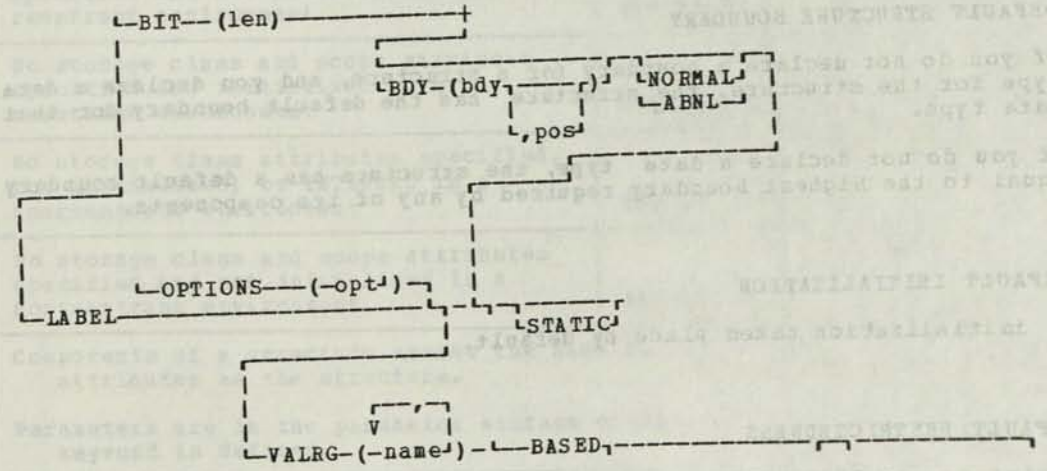
When a loop shows that a parameter can be repeated

(2)



When a line returns to the left of the diagram

If no left to right path exists for a combination of attributes, they are incompatible and may not be coded together. For example in the following excerpt from Figure 30:

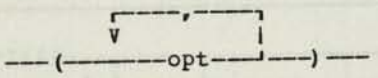


...BIT and VALRG are incompatible. The only path joining them goes from right to left on the horizontal line that joins LABEL and STATIC.

Bottom to Top Path Restrictions

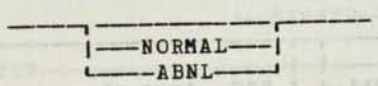
A bottom to top path may be taken in only two cases:

(1)



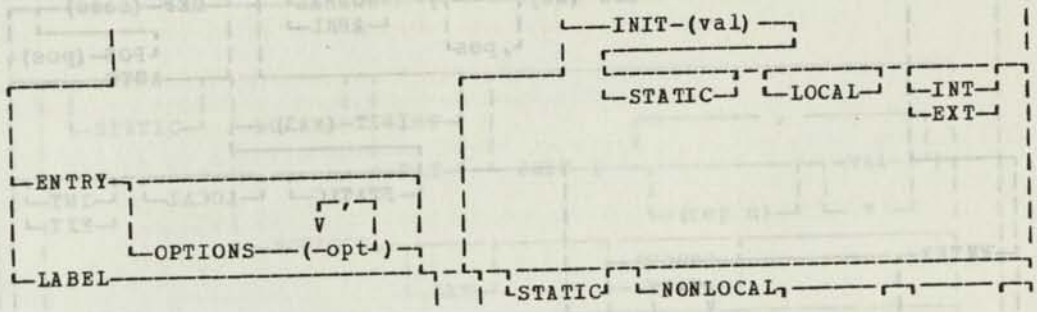
When a loop shows that a parameter can be repeated

(2)



When you have a return path to the "main line" from an option you dipped down to.

If no top to bottom path exists for a combination of attributes, they are incompatible and may not be coded together. For example, in the following excerpt from Figure 30:



...the attributes LABEL and INIT are incompatible. The only path joining them is from bottom to top between the horizontal lines after LABEL and before INIT.

NOTES ON THE COMPATIBLE ATTRIBUTES FIGURES

1. A component may not have a dimension attribute if a containing structure is dimensioned.

2. A component may not have the INITIAL attribute if a containing structure has been initialized, or if the containing structure has been declared other than STATIC LOCAL.

3. For descriptions of the attributes, see the description of the DECLARE statement in Chapter 2 of this manual.

4. "len" means a declared length. "p" means a declared precision.

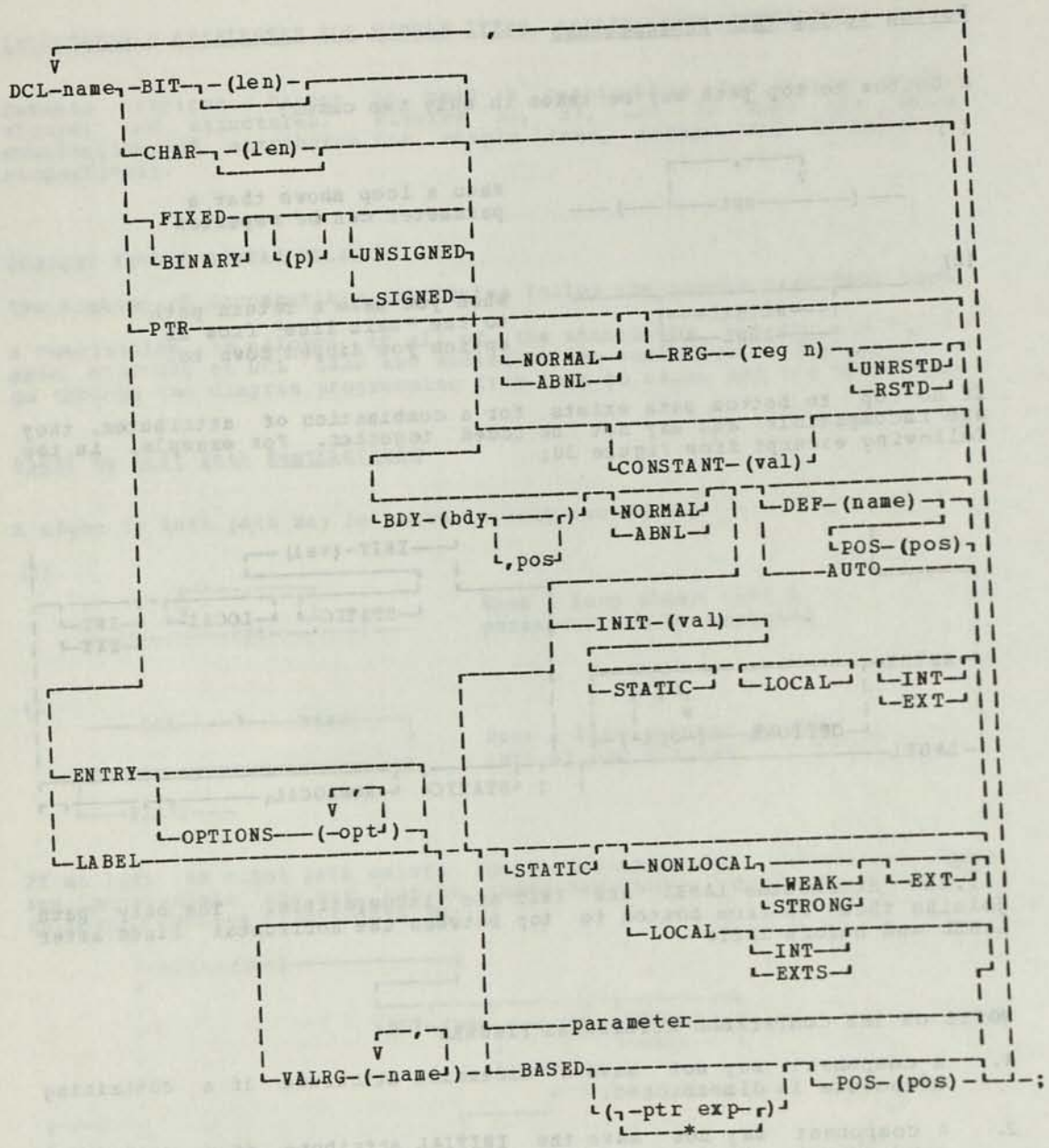
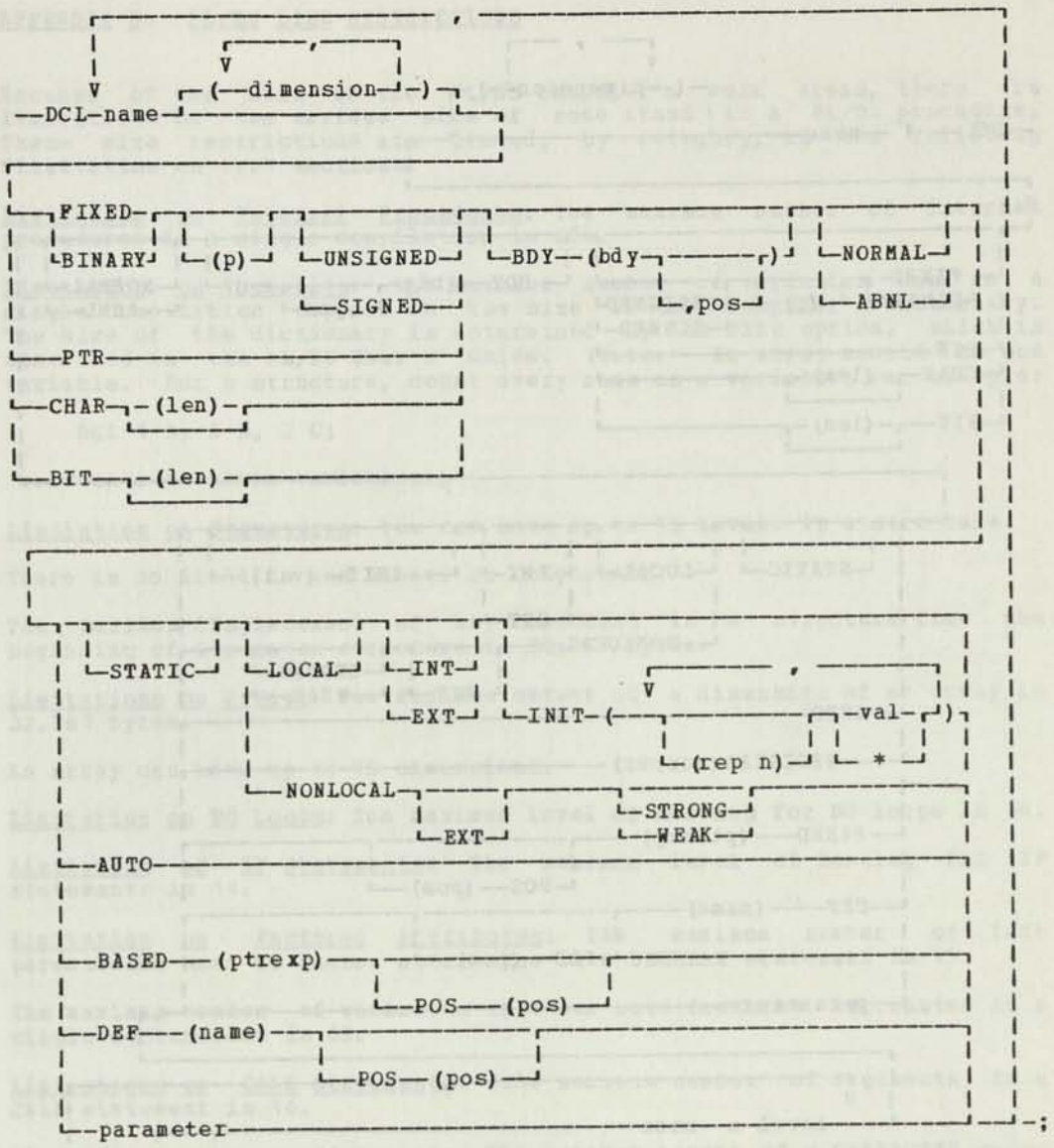


Figure 30. Compatible Attributes for Simple Items



Note: The repetition in INIT is valid only if the name has dimension.

Figure 31. Compatible Attributes for Arrays

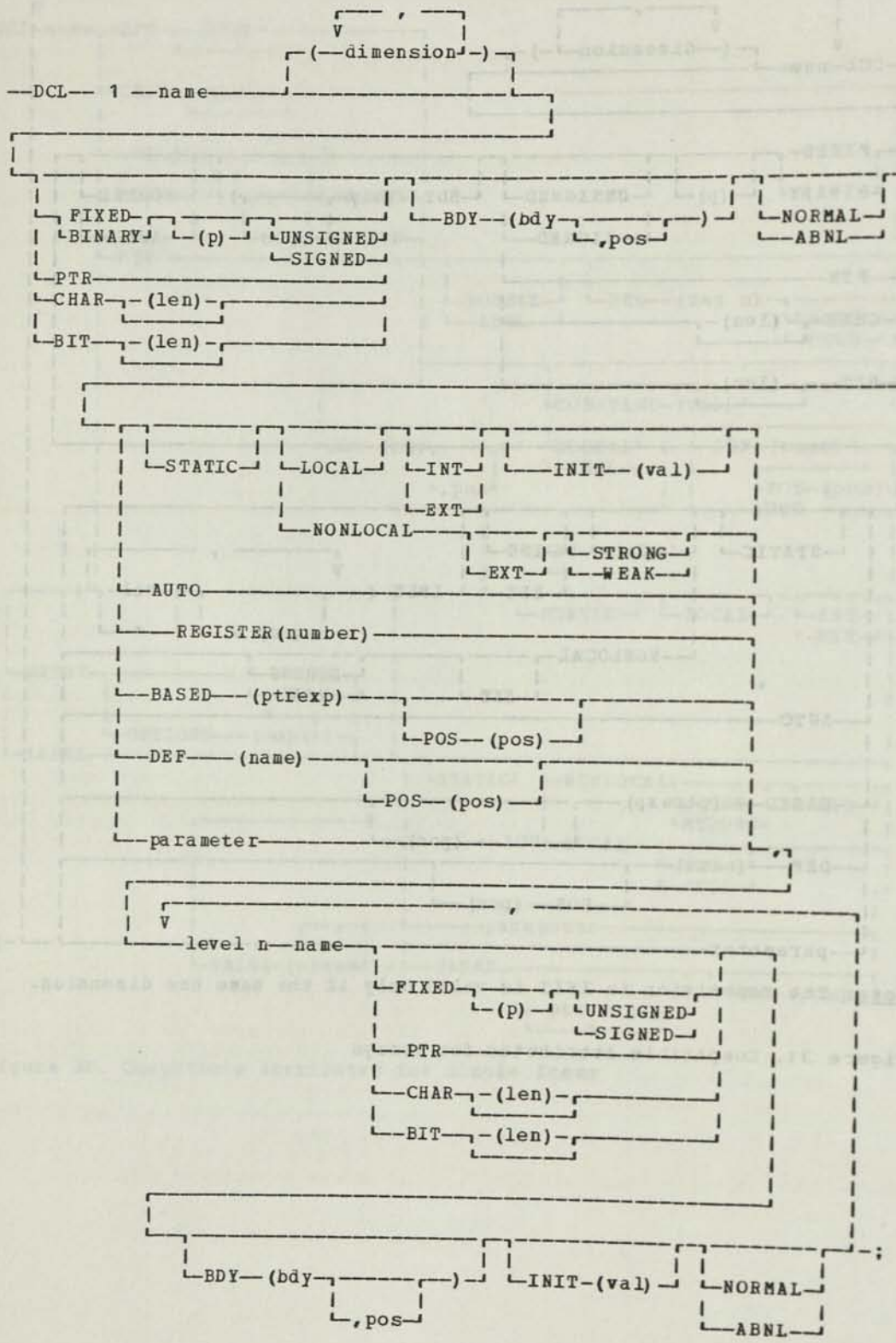


Figure 32. Compatible Attributes for Structures

APPENDIX B. PL/DS SIZE RESTRICTIONS

Because of the size of the PL/DS compiler's work areas, there are limitations on the maximum size of some items in a PL/DS procedure. These size restrictions are listed, by category, in the following "Limitation on ..." sections.

Limitation on Internal Procedures: The maximum number of internal procedures in a single compilation is 254.

Limitation on Variables: The maximum number of variables used in a single compilation depends on the size of the compiler's dictionary. The size of the dictionary is determined by the SIZE option, which is described in the PL/DS User's Guide. (Note: An array counts as one variable. For a structure, count every name as a variable; for example:

```
DCL 1 A, 2 B, 2 C;
```

...counts as three variables.)

Limitation on Structures: You can have up to 15 levels in a structure.

There is no limit to the number of components.

The maximum displacement of any component in a structure from the beginning of its major structure is 32,767 bytes.

Limitations on Arrays: The maximum extent of a dimension of an array is 32,767 bytes.

An array can have up to 15 dimensions.

Limitation on DO Loops: The maximum level of nesting for DO loops is 14.

Limitation of IF Statements: The maximum level of nesting for IF statements is 14.

Limitation on Factored Attributes: The maximum number of left parentheses used to factor attributes on a DECLARE statement is 15.

The maximum number of variables that can have factored attributes in a single declaration is 63.

Limitations on CALL Statements: The maximum number of arguments in a CALL statement is 16.

Limitations on Macro Statements: The maximum length of a CHARACTER macro variable is 1000 bytes. The total length of all CHARACTER macro variables should not exceed 45,000 bytes at any one time.

The range of a FIXED macro variable is from -2147483647 to 2147483647.

Limitation on Parameters: The maximum number of parameters passed to an entry point is 16.

Limitation of DEFINED Items: The maximum number of levels of DEFINED items (i.e. items DEFINED on other DEFINED items) is 15.

APPENDIX C. PL/DS LANGUAGE KEYWORDS

The following lists all the words recognized by the PL/DS compiler as language keywords. Those keywords that are reserved cannot be used as variable names.

Keyword	Permissible Abbreviation	Reserved
ABNORMAL	ABNL	
ABS		
ADDR		
AUTODATA		
AUTOMATIC	AUTO	
AUTOREG		
BASED		
BINARY	BIN	
BIT		
BOUNDARY	BDY	
BY		*
BYTE		
CALL		YES
CHARACTER	CHAR	
CONSTANT		
DECLARE	DCL	YES
DEFINED	DEF	
DIM		
DO		YES
ELSE		YES
END		YES
ENDGEN		YES
ENTRY		YES
EVAL		
EXIT		
EXTERNAL	EXT	
FIXED		
FLOWS		
GOTO	GO TO	YES
HWORD		
ID		
IF		YES
INITIAL	INIT	
INTERNAL	INT	
LABEL		
LENGTH		
LOCAL		
LINKAGE		
MACGEN		YES
MAIN		
MAX		
MIN		
NOAUTODATA		
NOAUTOREG		
NOEXIT		
NOFLOWS		
NOID		
NONLOCAL		
NOREFS		
NORMAL		
NOSAVE		
NOSTATNAME		
NOSTATORG		
NOSEQFLOW		
NOSETS		
NOSTATREG		
NOTEMPS		
OPTIONS		
POINTER	PTR	
POSITION	POS	
PROCEDURE	PROC	YES
REENTRANT		
REFS		
REGISTER	REG	
RESPECIFY	RFY	YES
RESTRICTED	RSTD	
RETURN		YES
RETURN TO		YES
SAVE		
STATNAME		
STATORG		
SEQFLOW		
SETS		
SIGNED		
STATREG		
STRONG		
STATIC		
TEMPS		
THEN		YES
TO		*
UNRESTRICTED	UNRSTD	
UNSIGNED		
UNTIL		*
VALUERANGE	VALRG	
WEAK		
WHILE		*
WORD		
WORKREGS		
NOWORKREGS		

* reserved following DO

Figure 33. PL/DS Language Keywords

APPENDIX D. PRECISION OF ARITHMETIC EXPRESSIONS

An arithmetic expression is two fixed variables separated by an arithmetic operator or sign. The following tables show the precision associated with arithmetic expressions. The columns represent the attributes of the right operands. The rows represent the attributes of the left operands. The intersection of a row and column shows the resulting precision.

If a given attribute does not appear in these tables, then that attribute cannot be used in an arithmetic expression; for example, "CHAR(5)" cannot be used in an arithmetic expression.

BIT items must be aligned on a byte boundary.

RIGHT OPERAND

LEFT OPERAND	FIXED (8)	FIXED (15)	FIXED (16)	FIXED (32)	CHAR (1)	CHAR (2)	BIT (8)	BIT (16)
FIXED(8)	15	15	16	32	15	16	15	16
FIXED(15)	15	15	16	32	15	16	15	16
FIXED(16)	16	16	16	32	16	16	16	16
FIXED(32)	32	32	32	32	32	32	32	32
CHAR(1)	15	15	16	32	15	16	15	16
CHAR(2)	16	16	16	32	16	16	16	16
BIT(8)	15	15	16	32	15	16	15	16
BIT(16)	16	16	16	32	16	16	16	16

Figure 34. Table of Precision of Add(+) and Subtract(-) Expressions

RIGHT OPERAND

LEFT OPERAND	FIXED (8)	* FIXED (15)	FIXED (16)	FIXED (32)	CHAR (1)	CHAR (2)	BIT (8)	BIT (16)
FIXED(8)	16	16	16	32	16	16	16	16
* FIXED(15)	16	16	16	32	16	16	16	16
FIXED(16)	16	16	16	32	16	16	16	16
FIXED(32)	32	32	32	32	32	32	32	32
CHAR(1)	16	16	16	32	16	16	16	16
CHAR(2)	16	16	16	32	16	16	16	16
BIT(8)	16	16	16	32	16	16	16	16
BIT(16)	16	16	16	32	16	16	16	16

* FIXED(15) is treated as FIXED(16)

Figure 35. Table of Precision of Multiplication (*) Expressions

RIGHT OPERAND

LEFT OPERAND	FIXED (8)	* FIXED (15)	FIXED (16)	FIXED (32)	CHAR (1)	CHAR (2)	BIT (8)	BIT (16)
FIXED(8)	16	16	16	error	16	16	16	16
* FIXED(15)	16	16	16	error	16	16	16	16
FIXED(16)	16	16	16	error	16	16	16	16
FIXED(32)	16	16	16	error	16	16	16	16
CHAR(1)	16	16	16	error	16	16	16	16
CHAR(2)	16	16	16	error	16	16	16	16
BIT(8)	16	16	16	error	16	16	16	16
BIT(16)	16	16	16	error	16	16	16	16

* FIXED(15) is treated as FIXED(16)

Figure 36. Table of Precision of Divide (/) Expressions

RIGHT OPERAND

LEFT OPERAND	FIXED (8)	* FIXED (15)	FIXED (16)	FIXED (32)	CHAR (1)	CHAR (2)	BIT (8)	BIT (16)
FIXED(8)	8	16	16	error	8	16	8	16
* FIXED(15)	8	16	16	error	8	16	8	16
FIXED(16)	8	16	16	error	8	16	8	16
FIXED(32)	8	16	16	error	8	16	8	16
CHAR(1)	8	16	16	error	8	16	8	16
CHAR(2)	8	16	16	error	8	16	8	16
BIT(8)	8	16	16	error	8	16	8	16
BIT(16)	8	16	16	error	8	16	8	16

* FIXED(15) is treated as FIXED(16)

Figure 37. Table of Precision of Remainder (//) Expressions

APPENDIX E. LINKAGE(3) OPTIONS FOR PROGRAMS RUN UNDER DPPX BASE

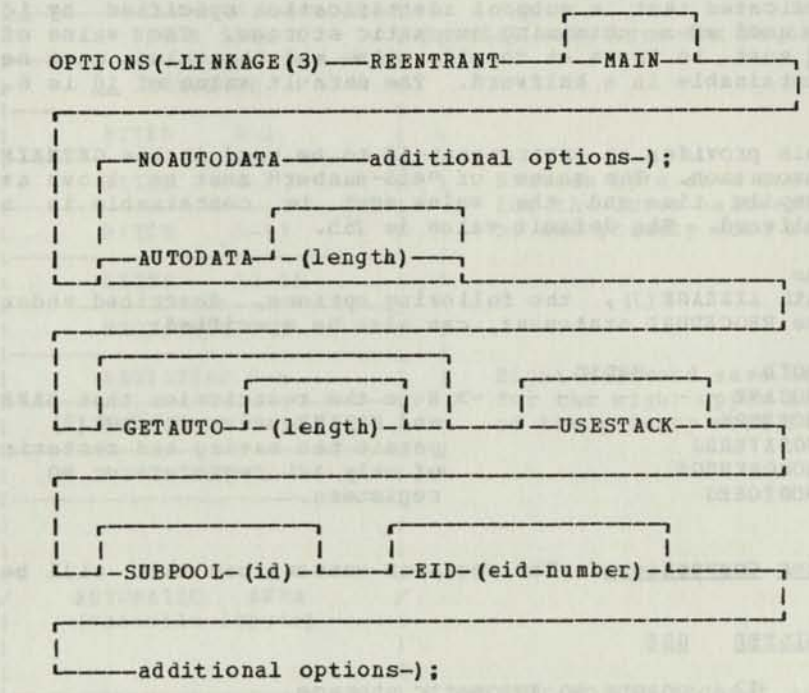
This section describes PROCEDURE statement options that can be used when the main PROCEDURE statement has

```
OPTIONS( LINKAGE(3) REENTRANT ...)
```

coded. The code sequences generated match the linkage protocols of the DPPX operating system.

The generated code depends on the fact that DPPX Base has always-resident modules to aid inter-program linkage.

The options can be coded as follows:



LINKAGE(3) Indicates that this procedure is to use the linkage conventions of DPPX Base.

MAIN Indicates that this procedure is invoked via supervisor-assisted linkage. This in turn implies that: the procedure will not save the registers in the prolog; and The procedure will perform a RETURN by invoking the EXIT function of DPPX Base.

NOAUTODATA Indicates that no automatic storage should be required by the procedure being compiled. This implies that there will be no request for automatic data nor will there be any adjustment of the stack pointer. An error message will be issued if the compiler is unable to satisfy this constraint.

AUTODATA(length) Indicates the maximum length permitted for automatic storage for the procedure being compiled. An error message will be issued if the compiler is unable to satisfy this constraint. If this option is not specified, the compiler will perform no checks about the amount of automatic data required by the compiled module.

USESTACK — Indicates that this procedure should attempt to acquire its automatic storage by decrementing the stack pointer instead of issuing a GETMAIN macro.

GETAUTO(length) — Indicates that the DPPX GETMAIN function can be used to acquire automatic storage. If length is omitted, then the actual amount of automatic storage needed by this compilation will be used. Typically the length specified is relatively large so that a GETMAIN is done to establish a stack of automatic data to be used by subsequently-called procedures. These subsequently-called procedures can then use the USESTACK option. Note: If USESTACK is coded in conjunction with this option, GETMAIN will not be invoked if there is sufficient space on the stack to supply the needed automatic storage.

SUBPOOL(id) — Indicates that a subpool identification specified by id be used when obtaining automatic storage. The value of id must be known at compile time and the value must be containable in a halfword. The default value of id is 6.

EID(eid-number) — This provides an environment-id to be used in the GETMAIN invocation. The value of "eid-number" must be known at compile time and the value must be containable in a halfword. The default value is 255.

additional options — With LINKAGE(3), the following options, described under the PROCEDURE statement, can alsc be specified:

ID	NOID	ENDID	
SAVE	NOSAVE	- - - - - - - - ->	Note the restriction that SAVE and NOSAVE under LINKAGE(3) permit the saving and restoring of only ALL registers or NO registers.
TEMPS	NOTEMPS		
NOSAVEAREA	NOSAVEREG		
WORKREGS	NOWORKREGS		
NORETREG	NORTOREG		

LINKAGE(3) Register Conventions: The register conventions that will be used are:

NAME	REGISTER	USE
AUTOREG	12	points to Automatic storage
BRANREG	08	Branch Address
RETREG	08	Return Address
RCODREG	30	Return Code Value
PARMREG	22	points to Parameter list
SAVEREG	12	points to byte at end cf caller's save area

Save Area Format: The format of the save area is as follows:

```
DECLARE
 1 SAVEAREA BASED BDY(WORD),
   3 CBF PTR,       /* POINTS TO HIGHER SAVE AREA    */
   3 SSP PTR,       /* POINTS TO START OF STACK      */
   3 *   CHAR(8),   /* RESERVED                      */
   3 QB  CHAR(16),  /* QUADRANT FOR LOWER HALFWORD
                       PORTION OF PRIMARY REGISTERS */
   3 QE  CHAR(16);  /* QUADRANT FOR UPPER HALFWORD
                       PORTION OF PRIMARY REGISTERS */
```

On entry, @SAVEREG points to the byte following this save area.

USESTACK Details: With USESTACK, the compiler assumes that @SAVEREG points to a save area as described above, and that the area behind the save area (i.e., has a lower logical address) is available for automatic storage for the procedure being ccmpiled. The start of stack pointer is used to check that sufficient space is available.

When GETMAIN is invoked (GETAUTO option), a new stack is created and the automatic storage is allocated starting from the highest logical address of the stack thus created.

The following shows the contents, in storage, of a "stack."

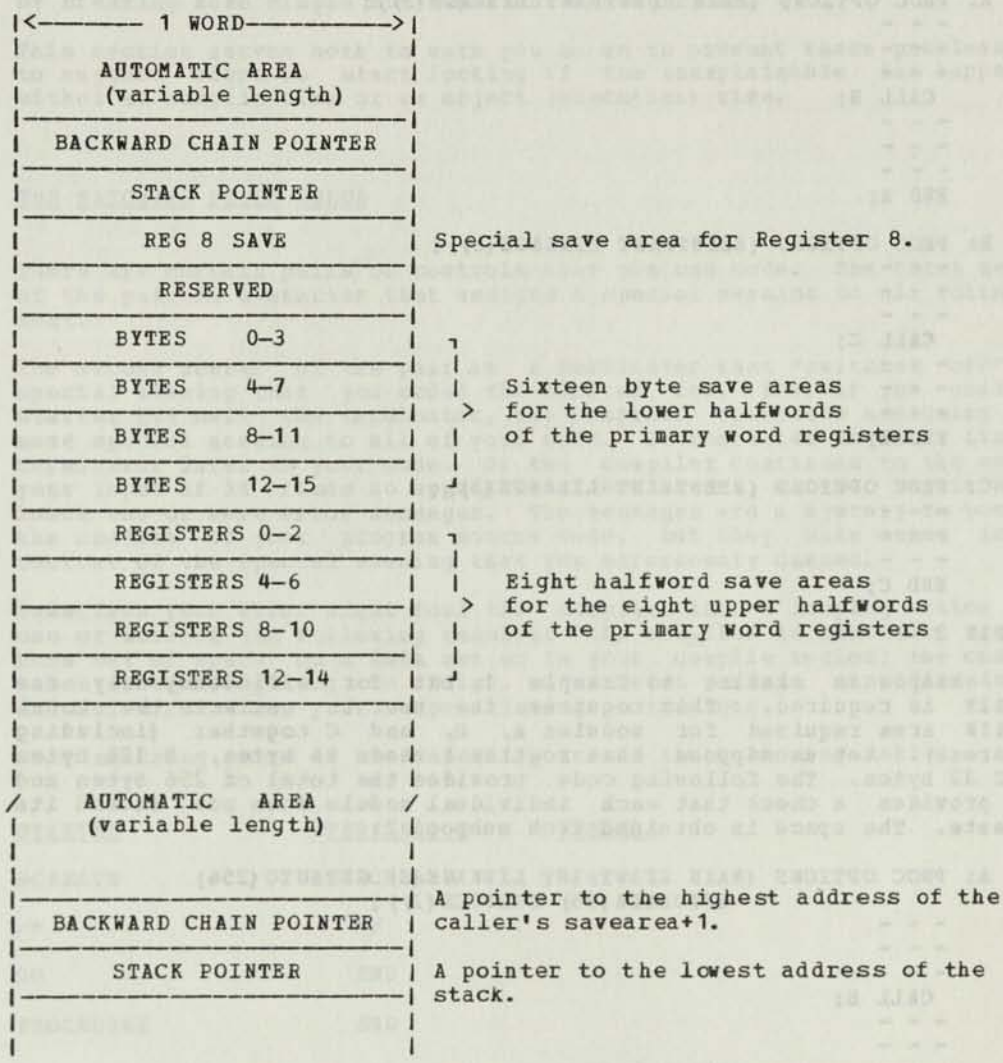


Figure 38. Format of a Stack

EXAMPLE 1

Routine A calls routine B, which in turn calls routine C. The three routines need to be reentrant. In this example the compiler issues a GETMAIN at the start of each routine and a FREEMAIN at the end of each routine. At the end of A (the main module), the compiler issues an EXIT macro to return to the supervisor.

```
A: PROC OPTIONS (MAIN REENTRANT LINKAGE(3));
   - - -
   - - -
   - - -
    CALL B;
   - - -
   - - -
   - - -
   END A;

B: PROC OPTIONS (REENTRANT LINKAGE(3));
   - - -
   - - -
   - - -
    CALL C;
   - - -
   - - -
   - - -
   END B;

C: PROC OPTIONS (REENTRANT LINKAGE(3));
   - - -
   - - -
   - - -
   END C;
```

EXAMPLE 2

This example is similar to Example 1, but for efficiency only one GETMAIN is required. This requires the user to estimate the total GETMAIN area required for modules A, B, and C together (including saveareas). Let us suppose that routine A needs 96 bytes, B 128 bytes and C 32 bytes. The following code provides the total of 256 bytes and also provides a check that each individual module does not exceed its estimate. The space is obtained from subpool 2.

```
A: PROC OPTIONS (MAIN REENTRANT LINKAGE(3) GETAUTO(256)
                 AUTODATA(96) SUBPOOL(2));
   - - -
   - - -
   - - -
    CALL B;
   - - -
   - - -
   - - -
   END A;

B: PROC OPTIONS (REENTRANT LINKAGE(3) USESTACK AUTODATA(128));
   - - -
   - - -
   - - -
    CALL C;
   - - -
   - - -
   - - -
   END B;

C: PROC OPTIONS (REENTRANT LINKAGE(3) USESTACK AUTODATA(32));
   - - -
   - - -
   - - -
   END C;
```

APPENDIX F. PREVENTION AND DETECTION OF UNEXPECTED COMPILER RESULTS

Occasionally you receive a compiler listing and find that some of it is far from what you expected, and that the error messages do not seem logical.

There are several ways you might unintentionally mislead the compiler, by breaking some simple but important coding rules.

This section serves both to warn you so as to prevent these problems and to suggest where to start looking if the unexplainable has happened, either at compile time or at object (execution) time.

THE MATCHING PAIRS ERROR

There are certain pairs of controls that you can code. The first member of the pair is a starter that assigns a special meaning to all following text.

The second member of the pair is a terminator that "switches off" the special meaning that you coded the starter for. Now, if you code the starter but omit the terminator, the compiler continues assigning this same special meaning to all of your code. It continues until it finds a terminator later in your code. Or the compiler continues to the end of your input if it finds no appropriate terminator. On the way it might issue one or more error messages. The messages are a mystery to you, in the context of your program source code, but they make sense in the context of the special meaning that you erroneously caused.

Sometimes your error might fool the compiler into a looping action with one or more of the following results: the compiler is cancelled when it runs out of space on a data set or in your compile region; the compile takes excessively long to finish, and is even perhaps cancelled due to excessive run time; your output listing is too long.

Some matching pairs to use carefully are in the following table.

STARTER	TERMINATOR	REMARKS
@CREATE	@ENDCREATE	
/*	*/	
DO	END	
PROCEDURE	END	
'	'	quoted strings
?(trigger character)	;	semicolon marks end of statement.
(	)	

ERROR LOOPS

An indication of incorrect code is an error loop, consuming excessive amounts of one or more system resources, as described above in the discussion of looping under "The Matching Pairs Error." Besides unusual interpretations of code due to missing terminators, already described, there are several coding errors that the compiler cannot correct, so you must be on the alert for them.

MACRO VARIABLE REPLACEMENT LOOP DURING COMPILE

In the macro processing phase, the compiler scans each piece of non-macro text and answer text for strings that match the names of macro variables. It replaces each such string (target variable) with the value assigned to the macro variable. (The macro variable must be activated and the answer text must have rescan in effect -- see the ACTIVATE and ANSWER statement.) It rescans what it replaced looking for any further matching macro variable. If, in your code, variable X is replaced by value X, or if variable X is replaced by value Y and variable Y is replaced by value X, your compile will have a replacement loop.

When the compiler detects that it has made an excessive number of consecutive replacements, it leaves the original target string with no replacements and issues an appropriate message.

EXECUTION TIME LOOPS

If there appears to be an unusual loop during actual use (execution) of a program -- especially if it is compiled with the optimize option -- see the discussion of optimization later in this appendix.

OPTIMIZATION CAUTIONS

The compile-time option "OPTIMIZE" causes a longer compile time, during which the compiler produces code that is significantly more efficient than code produced under the NOOPTIMIZE option. Optimized and unoptimized code are equivalent; that is, given the same input data, the execution of the optimized and unoptimized versions yields the same output. However, the optimized version requires fewer bytes and/or fewer execution cycles than the non-optimized version. This lets you trade off host compile time against execution time on the 8100.

It is possible for the optimized version of some programs tc produce different output than the nooptimize version of the same program. There are two possible reasons for the failure to have equivalence between optimized and non-optimized versions:

(1) The code being compiled has stringent design criteria that can only be satisfied by compiling with the OPTIMIZE option (REENTRANT code with the NOAUTODATA option frequently has this characteristic).

(2) The compiler's optimization phase performs transformations that "change the meaning of the program." The compiler might perform such transformations when you have failed to supply some information to the compiler.

You must supply information about:

- Register restrictedness
- Variables being changed
- Program flow

These three topics are discussed here.

REGISTER RESTRICTEDNESS AS AN OPTIMIZATION PROBLEM

The compiler may, at any statement, generate code that uses any unrestricted register. You cannot predict which register the compiler will use. If optimization is run, a given register may be reserved by the compiler, to hold the value of some expression. Sometimes this internal reservation of registers is in effect over most or all of the compiled code.

Sometimes a given source statement changes the value of a specific register, but there is no way that the compiler can determine this. You must inform the compiler of your restricted registers before you use them:

```
RESPECIFY (register-list) RESTRICTED;
```

Note: Over-restriction of registers can lead to sub-optimal code when you no longer need your restricted registers; free them them up by coding:

```
RESPECIFY (register-list) UNRESTRICTED;
```

Failure to restrict register(s) can lead to the generation of invalid code.

The statements that can potentially cause trouble are:

- CALL to an external routine
- MACGEN
- Control Immediate (KI) supported machine instruction

These problems are described below.

Register Restrictedness at a CALL Statement

Obviously, a CALL statement does not preserve the values of BRANREG, RETREG, and PARMREG. Other registers, however, are not necessarily destroyed. Therefore, the compiler might generate code that assumes the register contents are preserved. If the called routine does not preserve the contents of some other register, then you must surround the CALL statement with;

```
RFY reg-list RSTD
```

...and

```
RFY reg-list UNRSTD;
```

statements.

When you specify LINKAGE(3) on the PROCEDURE statement, the compiler will make additional assumptions. In this case, the compiler-generated code will be based on the assumption that a CALL-external statement will change BRANREG, RETREG(8), PARMREG(22), and registers 16, 24, 26, 28 and 30. This conforms to the linkage conventions of DPPX.

Register Restrictedness in MACGEN and Control Immediate Instructions

The compiler-generated code assumes that a MACGEN does not corrupt any register. It is a user responsibility to surround the MACGEN with the appropriate RFY statements.

In practice, this responsibility usually resides with the author of a macro that includes a MACGEN as part of its answer text. Also note that in practice, a MACGEN is apt to have the same register preservation characteristics as a CALL, but this assumption is not part of the language.

A few control immediate instructions can change all secondary or primary registers, because they change the active register space. This fact is not "known" to the compiler and therefore you must again use RFY statements, as described above. In practice, very few programmers need be concerned with this point.

CHANGING OF VARIABLES AS AN OPTIMIZATION PROBLEM

This is the second of three potential sources of optimization problems. A significant feature of optimization is that it may keep history on expressions (e.g., "A+B", "P->Q", "ADDR(X(I))"). The value of such an expression may be carried in a register. Therefore, the compiler does not repeatedly reissue the code to re-evaluate the expression. Whenever you change one of the contributors to a given expression, however, the compiler must re-evaluate the expression at its next occurrence.

The compiler can detect that a variable has changed when one of the following occurs:

- You code the variable as the receiver of an assignment statement.
- You code the variable as the control variable of a do-loop.
- You code the variable as an output argument of a supported machine instruction.

It is possible for a variable to change through the action of a CALL-external statement. The compiler makes certain assumptions at a CALL. These are: formal parameters will be changed by the called routine; restricted registers will be changed; EXTERNAL variables will be changed; and variables BASED (constant) will be changed (this is similar to EXTERNAL data). On some occasions some other variable might be changed. In this case, you must code the SETS option on the DECLARE of the called ENTRY.

The compiler makes no assumptions about a MACGEN statement; you must properly code the SETS option on the MACGEN statement. (In practice, this is the responsibility of the author of a macro.)

Indiscriminate use of pointers can introduce errors. The compiler generates code based on the assumption that any two different pointers will never point to the same area of real storage. The compiler makes no attempt to trace the assignments of values to pointers to detect when a subtle form of aliasing is in effect.

Consider the following example:

```
1.  Z = P->X;      /* FIRST USE OF "P->X" CAUSES REG TO CONTAIN VALUE */
2.  Q = P;
3.  Q->X = 0;      /* ACTUALLY CHANGES "P->X" BUT NOT DETECTED        */
4.  W = P->X+1;    /* MIGHT ERRONEOUSLY RE-USE "P->X"                 */
```

In the above example line 3 changed "P->X" but this is not detected and the compiler might (erroneously) keep "P->X" in a register from line 1 to line 4.

You can avoid this problem with one of the following:

- Rewrite the code to use a single pointer name
- Insert a statement of the form:

```
P=P;
```

...This will make optimization reevaluate any expression based on "P."

- Insert a MACGEN statement with the appropriate SETS information. For example:

```
MACGEN SIZE(0) SETS(P);
```

PROGRAM FLOW AS AN OPTIMIZATION PROBLEM

This is the third of three areas of potential optimization problems.

The compiler must be aware of any possible flow of control during execution of the procedure it is compiling. This is automatic when you use structured code (such as, IF...THEN...ELSE), but there are several conditions that the compiler is unable to analyze. You must supply additional information in the following cases:

DECLARE of BASED LABELs (see VALUERANGE)

DECLARE of BASED ENTRYs (see VALUERANGE)

Out-of-line return from CALLed entry (see DECLARE of ENTRY)

Out-of-line flow from MACGEN (see FLOWS keyword of MACGEN)

IF YOU SUPPLY INCOMPLETE OR INCORRECT INFORMATION, INVALID CODE MAY RESULT.

UNEXPECTED RESULTS WHEN USING THE RESPECIFY STATEMENT

Your program may give unexpected results if you incorrectly position RESPECIFY statements with the RESTRICTED or BASED options in your source code.

You should look upon RESPECIFY, with the RESTRICTED or BASED options, as an instruction to the compiler. It is not execution time logic.

During the compile phase, the compiler implements immediately each RESPECIFY statement with the RESTRICTED or BASED option. Every statement that follows it has the result of the RESPECIFY compiled into it.

If, at execution time your logic bypasses that part of your code where you had the RESPECIFY statement, it makes no difference. The RESPECIFY is in effect anyway. This is similar to the DPPX Assembler USING command.

See the example below. Statement 6 is a RESPECIFY ... BASED instruction. Statement 8 is a RESPECIFY ... BASED instruction. The branch from statement 7 to statement 12 bypasses statement 8. However, because the compiler implements statement 8 at compile time, you would be wrong to assume that:

(a) because statement 6 is the last statement in which you respecified X in your logic flow, then X is based Q in L2 and L3.

(b) because your logic flow bypassed statement 8, then the assignment X based P is not in effect.

```
1.    DCL X BASED(P),
2.    (P,Q) PTR;
3.      - - -
4.      - - -
5.    X=0;          /*  EQUIVALENT TO P->X=0  */
6.    RFY X BASED(Q);
7.    GO TO L1;
8.    RFY X BASED(P);
9.    L2: RETURN;
10.     - - -
11.     - - -
12.   L1: X=0;      /*  EQUIVALENT TO P->X=0, NOT Q->X=0  */
13.   GO TO LZ;
```

COMMON PROBLEMS

This section attempts to answer questions that are frequently asked by PL/DS users. Items of language that are frequently misunderstood are clarified here.

REGISTER Declares and MACGEN

The PL/DS compiler requires:

1. Register declarations must either be included (%INCLUDE) from a macro library or in answer text produced by invoking a macro definition that resides in a macro library.

2. The MACGEN statement must be in answer text produced by invoking a macro definition that resides in a macro library.

Example:

```
@PROCESS MACRO;
MAIN:PROC OPTIONS(...
     - - -
     - - -
     @INCLUDE SYSLIB(REGDCLS); /* INCLUDE REGISTER DECLARES  */
(or  %INCLUDE SYSLIB(REGDCLS); /* INCLUDE REGISTER DECLARES  */)
     - - -
     - - -
     - - -
     ?LINK EP(FRED);                /* PL/DS MACRO THAT RETURNS
                                       A MACGEN STATEMENT        */
END;
```

Message: IRE4701I T INSUFFICIENT REGISTERS ARE AVAIL ...

This terminating message is generated by the compiler if there are no further registers available for the code generation for a particular statement. This may be due to one of the following two reasons:

Reason 1: The statement is too complex to compile with the registers available at that statement. If the statement can be simplified then this should be tried first. A statement such as a variable length string assignment requires a number of work registers. Any expressions in the substrings should be assigned to temporary halfword variables and these used in the assignment statement.

Reason 2: Too many registers are restricted by the user at that statement. Register restrictedness should be checked and registers unrestricted if possible. See "Determining Register Restrictedness," in Appendix G. It may be that there are several registers available but not enough of a particular type; for example, too few primary registers.

Not All Local Variables Assigned to Registers With NOAUTODATA

If reentrant code is required and the linkage scheme does not provide any AUTOMATIC storage then it is necessary for all LOCAL variables to be assigned to registers. If a particular variable is not assigned to a register by the compiler this may be for one of the reasons itemized under "Ensure Local Variables Can Be Globally Assigned to Registers."

If none of these is the case then there are probably not enough registers available for this particular variable to be assigned. Following are approaches that may help to correct the problem.

Approach 1: Ensure register "restrictedness" is correct, that is, that there are no places where a register is restricted when it need not be.

Approach 2: Use different names for different uses of the same work variable. For example, use 'I1', 'I2'... rather than 'I' throughout.

Approach 3: Reestablish local pointer values if there are sections of the program where the pointer is not used. For example:

```
P=ADDR(...);                    /* P SET HERE */
- - -
'P' referenced in this section (a).
- - -
- - -
'P' not referenced in this section (b).
- - -
- - -
'P' referenced in this section (c).
- - -
```

Although P is not referenced in section (b) of the program, it is tying up a register, since its value is required later in section (c). Inserting another 'P=ADDR(...);' statement in front of the last section will free up that register for another use in the central section. If possible, use different pointers 'P1', 'P2', etc.

Approach 4: Change the linkage scheme so that more registers are saved/restored, thus freeing more registers for global assignment.

Approach 5: See whether some local variables can be eliminated completely. For example:

```
DCL RATIO   FIXED(16);
RATIO=X/Y;
IF RATIO > Z THEN...
```

...could be rewritten:

```
IF X/Y > Z THEN...
```

...thus eliminating the need for the variable 'RATIO'.

Approach 6: Ensure KIs, IOs, IOHs, and IOIs are specified correctly. (See, "Specify KI, IO, IOH, and IOI Instructions Correctly.")

Approach 7: Try using the WORKREGS procedure option. (See, "The WORKREGS Procedure Option," in Appendix G.)

Fullword/Halfword Arithmetic

The arithmetic precision rules in PL/DS state that if two halfword variables are added, subtracted, etc., the result is halfword. The following example shows the use of an intermediate assignment to obtain a fullword result.

```
*DCL X       FIXED(32),
*    (I,J)   FIXED(16);

*X=I+J;
          LHR   X,I
          AHR   X,J          (I+J IN HALFWORD X)
          XHR   @02H,@02H
          LHRE  X,@02H       (EXTENSION OF X ZERO)
```

If it is known that the addition of 'I' and 'J' could carry into the high-order halfword then the statement should be coded:

```
*X=I;
          XHR   X,X
          LHRE  X,X
          LHR   X,I
*X=X+J;
          AHR   X,J
          AYHRE X,0
```

...word addition takes place in the second statement because of the rule that if either of the operands is FIXED (32) then the result is FIXED(32).

Similarly, for a word result of a halfword multiply, an intermediate assignment must be made:

```
DCL X FIXED(32);
DCL (I,J) FIXED(16);
X=I*J;          /* HIGH-ORDER HALFWORD OF X WILL BE ZERO */

X=I;
X=X*J;          /* WORD*HALFWORD MULTIPLICATION          */
```

Message: IRE0612I I VARIABLE xxxx MAY BE USED BEFORE BEING SET

This message is generated if the compiler detects a situation where a local variable could potentially be referenced before having been initialized. For example:

```
T:PROC(A);
  DCL (X,Y,A);
  IF A=0 THEN
    Y=1;
  X=Y;
```

...produces the message since when A is not zero, Y is not set.

Sometimes it may be difficult to find whether there is a potential error or whether the compiler is being over-cautious. Following are examples of such cases:

Example 1:

```
DO I=A TO B;
  - - -
  Y=....;                /* Y SET        */
  - - -
END;
X=Y;                     /* Y REFERENCED */
```

The message is produced since the compiler has to assume that the loop is not executed when A has a value greater than B.

Example 2:

```
IF FLAG='1'B THEN
  Y=...;                 /* Y SET ONLY WHEN FLAG='1'B        */
  - - -
  - - -
IF FLAG='1'B THEN
  X=Y;                   /* Y REFERENCED ONLY WHEN FLAG='1'B */
```

Here, the user knows that Y will be set and referenced only when FLAG='1'B. However the compiler cannot detect that this is the case, and it issues the message.

Example 3:

```
DCL 1 FLAGS,
      2 FLAG1 BIT(1),
      2 FLAG2 BIT(1),
      2 FLAG3 BIT(1);
FLAG1='0'B;
FLAG2='0'B;
FLAG3='0'B;
```

...would produce the message VARIABLE "FLAGS" MAY BE USED BEFORE BEING SET even though all the subcomponents of the structure have been set. The message would not have been produced if the structure had been initialized:

```
FLAGS=''B;  /* INITIALIZE LOCAL FLAGS */
```

Note: When this message is produced, modify the program to eliminate it. If necessary, insert a statement to initialize the variable at a suitable point. Failure to do this inhibits optimization since, on entry to the procedure, the compiler assumes the variable is busy, and ties up a register unnecessarily.

APPENDIX G. PL/DS TECHNIQUES AND AIDS

SOME HINTS ON SOURCE CODE STYLE

The format of each procedure -- the way your source statements are set up -- can facilitate analyzing and modifying the code. We suggest the following conventions.

1. Avoid placing many PL/DS statements on a single line. This can cause difficulty when it is necessary to change one statement on the line.

2. Adopt specific conventions about the placement of comments in your source statements.

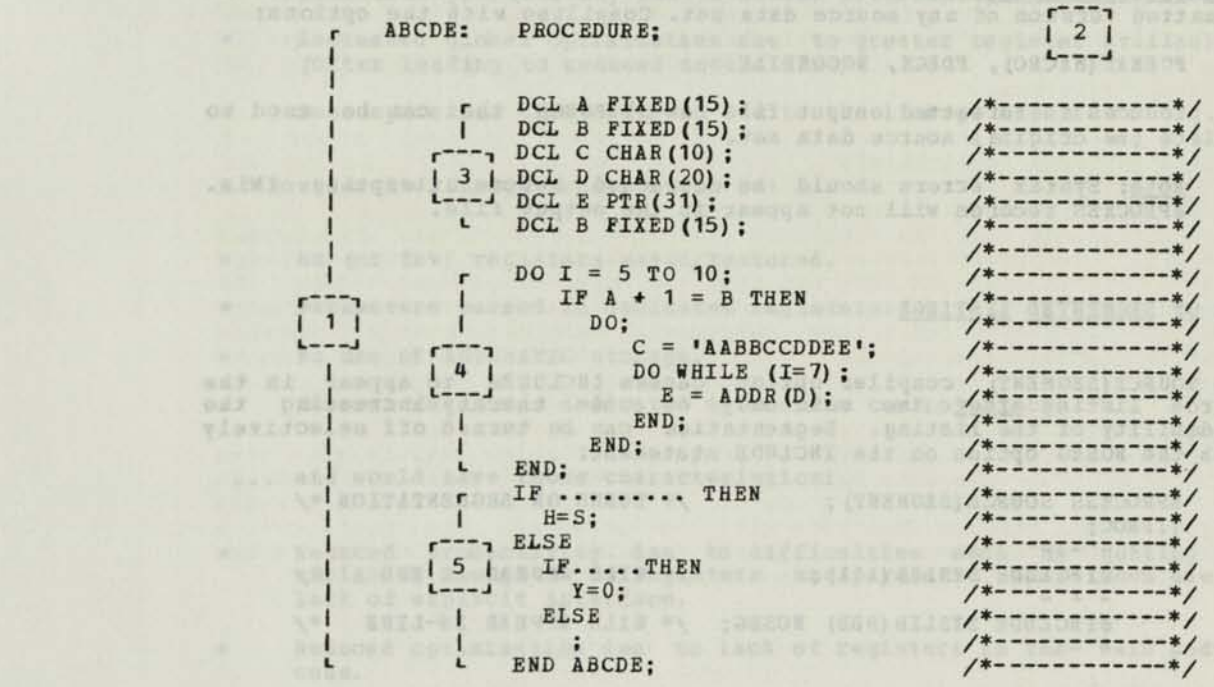
3. Group your DECLARE statements at the beginning of the procedure.

4. Indent each DO statement in a nest of DO statements, and line up each END statement under its associated DO statement. This ensures that you properly close each do-group. Also, indent and align all statements applicable to each DO statement.

5. For an IF statement, the IF and ELSE keywords should start in the same column on their respective input lines. When you write nested IF statements, the statements at the same nesting level should have the same starting columns.

Note: The FORMAT compiler option will assist in achieving these conventions. (See, "Compiler Options and Controls" and the PL/DS User's Guide.)

The following illustrates each of the formatting conventions listed above.



CONTROLLING ASSEMBLER CODE ANNOTATION

The ANNOTATE compiler option inserts the PL/DS source code as comments on the assembly listing (ALIST). Annotation does not begin until a statement that is not one of the following is encountered:

```
PROCEDURE  DECLARE  RESPECIFY  MACGEN
```

To ensure that declarations appear on the listing, add a null statement after the external PROCEDURE statement. For example:

```
@PROCESS NOM;
T:PROC;
; /* NULL STATEMENT TO CAUSE ANNOTATION OF DECLARES */
     DCL X FIXED(15),
         Y FIXED(15),
          - - -
          - - -
         ..........;
```

USES OF THE FORMAT OPTION

Comment Formatting: The PL/DS formatter has conventions for formatting comments that are coded in separate records. The column that the opening /* of the comment is in determines the formatter output of the comment, as follows:

If /* is in Column:	...the Format of the Comment is:
2,3, or 4	Expanded to whole line, with */ at right margin.
Column 5 or more	Enclosed in a box of /*, * and */ at current nesting level of formatted code.

Data Set Formatting: You can use the formatter (with care) to obtain a formatted version of any source data set. Compiling with the options:

```
FORMAT(MACRO), FDECK, NOCOMPILE
```

...produces a formatted output file on SYSPUNCH that can be used to replace the original source data set.

Note: Syntax errors should be corrected before attempting this. @PROCESS records will not appear in the output file.

USE OF SEGMENTED LISTINGS

The SOURCE(SEGMENT) compiler option causes INCLUDEs to appear in the source listing after the main body of code thereby increasing the readability of the listing. Segmentation can be turned off selectively with the NOSEG option on the INCLUDE statement:

```
@PROCESS SOURCE(SEGMENT);          /* TURNS ON SEGMENTATION */
T:PROC;
     - - -
     @INCLUDE SYSLIB(AAA);         /* WILL APPEAR AT END    */
     - - -
     @INCLUDE SYSLIB(BBB) NOSEG;   /* WILL APPEAR IN-LINE   */
     - - -
```

PERFORMANCE HINTS

This section contains hints for obtaining efficient generated code from the PL/DS Compiler without coding at a low assembler-style level.

Linkage Design

The importance of linkage design should not be underestimated. It can have an overriding effect on the performance of a system or application. A poorly thought-out linkage scheme also reduces programmer productivity by introducing extra problems, such as difficulty in getting all variables assigned to registers.

"Linkage" is used here to mean the way in which modules pass data and control between each other. The compiler provides various linkage options (described in the main chapters of this manual) but the way in which these options are used determines how effective the chosen linkage scheme will be. PL/DS gives the user freedom to choose his own level of control over linkage.

A "high-level" linkage scheme might be as follows:

- All registers saved and restored.
- Data communicated using formal parameters.
- Stack scheme used for AUTOMATIC storage.
- No dedicated registers for communication.

... and would have these characteristics:

- Increased productivity due to easier programming and clearer code.
- Increased global optimization due to greater register availability (often leading to reduced module size).
- Some space and time overhead due to the linkage itself.

A "low-level" linkage scheme might be:

- No (or few) registers saved/restored.
- Parameters passed in dedicated registers.
- No use of AUTOMATIC storage.
- Several registers dedicated to address control blocks.

... and would have these characteristics:

- Reduced productivity due to difficulties such as getting all variables assigned to registers and parameter mismatches due to lack of explicit interface.
- Reduced optimization due to lack of registers in the main body of code.
- Efficient code for the linkage itself.

In between these two extremes there are many different compromises available and the final choice will be determined by the system/application constraints.

Keep the following factors in mind while making the final choice of linkage design.

Register Usage: If only a few registers are available (that is, UNRESTRICTED) in the main body of a module, then the scope for optimization is reduced and the size of the generated code will be larger than necessary since expressions, address calculations, and other such things may have to be re-evaluated many times due to lack of registers to hold temporary values.

The number of available registers is increased if:

a. More registers are saved on entry and restored on exit from the procedure (and this can be relied upon when other procedures are CALLed).

b. Fewer registers are RESTRICTED in the program.

The Use of AUTOMATIC Storage: Overall storage savings can often be obtained by making modules reentrant and allowing the compiler to use AUTOMATIC storage for local variables that are not globally assigned to registers.

The size of the generated code for a program is minimized if the best possible use is made of registers. It is often better if registers can be used to hold expressions rather than local variables where "expressions" includes addresses, based variables, etc. However, if expressions are to be assigned to registers in preference to local variables there must be somewhere to store the local variables. If there is no storage available for local variables, then expressions may have to be re-evaluated many times.

The Use of Formal Parameters: The use of formal parameters should be considered with PL/DS where parameters (and their addresses) are considered as candidates for global register assignment along with other variables. Programs using formal parameters are often much easier to debug and maintain than those that pass data in control blocks or dedicated registers.

Avoiding Register Declarations

Variables should not be declared as registers unless absolutely necessary. The compiler is designed to assign to registers those variables and expressions that minimize the size of generated code. If a user declares a particular variable as a register in order to get more efficient code at a particular statement, this may have an adverse effect on optimization of the program as a whole. In general, it is better to leave the task of register allocation to the compiler.

Remember that it is not necessary for operands of most built-in machine instructions (such as KI) to be register variables (there are some exceptions). The compiler will generate housekeeping instructions to load and store operands if necessary. If the operands are in suitable registers then one-for-one code will be produced. The optimization phase of the compiler will, where feasible, automatically assign arguments of built-in instructions to registers of the proper type.

Choice of Registers Declared

We have recommended above that register declarations be avoided. However, if it is necessary to declare a register for a particular purpose then care should be taken when choosing which register to be used. On the 8100 some registers are more "useful" than others and the compiler will take account of this when assigning variables to registers. The user should try to use the same principles when choosing a particular register for a variable. Following is a summary of the 8100 registers and their "usefulness":

Base Registers	When one of the registers 12, 14, 28, or 30 is the base for a control block, then the two-byte LHS/STHS instructions can be used to address variables in the first 64 bytes. Hence, base registers are most suitable for basing small control blocks.
Registers 2,3	If registers 2 or 3 contain flags then if individual flags are tested, the JBZ (Jump Bit Zero) instruction can often be used. These registers are very useful for holding flags.
Primary Registers	Arithmetic and logical operations are more efficiently performed on primary registers (i.e. registers 0-15).
Secondary Registers	It is usually a good practice to use secondary registers (16-30) to hold addresses so that primaries are freed up for arithmetic operations. Logical operations will be less efficient when performed on secondary registers since the register will be loaded into primary, operated on, and then loaded back into secondary.

If a variable is declared as a register it should be declared UNRESTRICTED, and respecified as RESTRICTED/UNRESTRICTED as necessary. In this way the compiler can make use of the particular register in the portions of the program where it is UNRESTRICTED.

Use of Internal Procedures to Decrease Object Code

The use of internal procedures can be a significant factor in reducing the overall size of the generated code. The time and space overhead of calling an internal procedure is small, making it worthwhile to create an internal procedure if even a small number of statements are found to be common. The use of an internal procedure, even if only invoked from one point, can greatly increase a program's readability.

Note: If the NOAUTODATA compiler option has been specified (means that AUTOMATIC data must not be generated) then an internal procedure cannot have formal parameters, since storage is required for the parameter address list.

Alignment of Control Block Data

Alignment of data is very important on the 8100 since code to load misaligned data into registers is inefficient. For example:

```
*DCL P          PTR;
*DCL 1 S BDY(HWORD) BASED,
*       2 *  FIXED(15),
*       2 A  CHAR(2),          /* A IS BDY(HWORD)    */
*       2 *  CHAR(1),
*       2 B  CHAR(2);          /* B IS BDY(HWORD,2) */
*RFY S BASED(P);
*DCL X CHAR(2);
*X=A;
          LHS   X,A(P)          (2 bytes)
*X=B;
          L     X+0,B(P)
          L     X+1,B+1(P)      (8 bytes)
```

Wherever possible, design control blocks in such a way that data items have their correct boundary; for instance, words on word boundaries, halfwords on halfword boundaries.

Particular attention should be paid to CHARACTER and BIT items, which are multiples of 2 bytes, since they do not default to BOUNDARY(HWORD). They should be declared with BDY(HWORD) wherever possible, even if they are in a structure that has BDY(HWORD), since items may be added to the structure at a later date that could shift the item off boundary.

Even for single byte items in a structure, better code will be generated if the boundary is known to be BDY(HWORD) or BDY(HWORD,2) rather than unknown; for example, BDY(BYTE). This is because the LHS instruction can often be used to load a single byte if the item's boundary is known at compile time. In the following example, items in this structure have an unknown boundary since the structure will default to BDY(BYTE):

```
DCL 1 S BASED,          /* DEFAULTS TO BDY(BYTE) SINCE NO
                           ELEMENTS REQUIRE ALIGNMENT     */
      3 A FIXED(8),
      3 B FIXED(8),
      3 C FIXED(8);
```

... adding BDY(HWORD) at the level 1 would greatly improve the code required to access any item in the structure.

Size of Control Blocks

If control blocks are kept to 64 bytes or less in length, there is a greater chance that the LHS/STHS instructions will be used to load/store halfword and byte items in the structure. However, if the last item in the structure is an array, then it will not make any difference if the array makes the structure longer than 64 bytes. For example:

```
DCL 1 S BASED BDY(HWORD),
      2 A(100),
      2 B,
      2 C;
```

... is less efficient than:

```
DCL 1 S BASED BDY(HWORD),
      2 B,
      2 C,
      2 A(100);
```

If a structure is longer than 64 bytes, then byte and halfword items should come before word items.

Making Array Multipliers Powers of 2 for Efficient Code

The multiplier for an array is the byte distance between two consecutive elements. Code to address the I'th element of an array will be more efficient if this multiplier is a power of 2, since a simple shift instruction can be used. For example:

```
/* THESE ARRAYS HAVE MULTIPLIERS THAT ARE POWERS OF 2: */
     DCL A1(10) FIXED(8);          /* MULTIPLIER = 1  */
     DCL A2(13) FIXED(16);         /* MULTIPLIER = 2  */
     DCL A3(5)  CHAR(16);          /* MULTIPLIER = 16 */
/* WHEREAS THESE HAVE MULTIPLIERS THAT ARE NOT: */
     DCL A4(10) CHAR(3);           /* MULTIPLIER = 3  */
     DCL A5(9)  CHAR(12);          /* MULTIPLIER = 12 */
```

Particular care should be taken with arrays of structures:

```
/* THE MULTIPLIER FOR THIS ARRAY OF STRUCTURES IS 6 */
     DCL SDIM      FIXED CONSTANT(10);
     DCL 1 S(SDIM) BDY(HWORD),
           3 A     FIXED(16),
           3 B     FIXED(16),
           3 C     FIXED(16);

/* WHEREAS IT IS 8 IN THIS EXAMPLE: */
     DCL 1 S(SDIM) BDY(HWORD) CHAR(8), /* LENGTH SPECIFIED TO
                                  ENSURE MULTIPLIER IS POWER OF 2 */
           3 A     FIXED(16),
           3 B     FIXED(16),
           3 C     FIXED(16);

/* ANOTHER ALTERNATIVE MIGHT BE: */
     DCL 1 S       BDY(HWORD),
           3 A(SDIM)  FIXED(16),
           3 B(SDIM)  FIXED(16),
           3 C(SDIM)  FIXED(16);
```

The code required to address A(I) in the second structure will be much more efficient than that required to address A(I) in the first. In general then, make arrays of structures have an element length that is a power of 2.

Note: The length of a structure element can be found in the attribute and cross-reference listing.

Use of the OPTIMIZE Option

NOOPTIMIZE is the default and means exactly what it says: no optimization. In particular, local variables will not be globally assigned to registers without OPTIMIZE, even if NOAUTOTDATA has been specified, and therefore assembler errors may result. However, it is good practice to compile with NOOPT until all syntax and semantic errors have been corrected.

The compiler supports OPT(SPACE) and OPT(TIME) with OPT(SPACE) as the default. The only difference is in the generated code for string padding. With OPT(TIME) an overlapping MVS instruction is used. It is faster, but takes more space than a BCTR loop.

Caution: Use OPTIMIZE with care. See the description of optimization techniques in Appendix F.

Ensure Local Variables Can Be Globally Assigned to Registers

A LOCAL variable is a candidate for global assignment to a register unless any one of the following is true:

- It is an item in a structure (and the whole structure cannot be contained in a register).
- It has the DEFINED attribute.
- Another item is DEFINED on it.
- It has the attribute ABNORMAL.
- It has the attribute EXTERNAL.
- It has the attribute INIT.
- It is CHARACTER and substringed.
- It is the argument of the ADDR built-in function.
- It is a CALL argument which is not within EVAL. For example:

```
CALL SUB(EVAL(X)); /* X is still a candidate */
CALL SUB(X);       /* X not a candidate      */
```

- It is an argument of a built-in instruction that implies the variable is in storage. For example:

```
LHS(Y,X);          /* X not a candidate      */
```

Avoid the above characteristics when declaring and using local variables, to ensure their global assignment to registers.

Specify Control Immediate and Input/Output Instructions Correctly

There are three categories of Control-Immediate (KI) operations:

1. Those that set the first operand. These should be specified as KIR (KI-Read).
2. Those where the first operand is an output field. These should be specified as KIW (KI-Write).
3. Those where the first operand is a dummy (for example, set master mask). These should be specified as KIZ (KI-Zero).

If KI alone is specified, then the compiler will assume the first operand to be both an input and an output field, which will sometimes cause redundant loads and stores to be generated. Even if the operand is a register and there are no extra instructions generated for that particular statement, overall optimization can be improved by correct specification of these instructions.

The same guidelines should be followed with the Input/Output instructions IO, IOH, and IOI.

FIXED(8) Versus FIXED(15)/FIXED(16) For Local Variables

There are more addressable register bytes than halfword registers on the 8100, and therefore there is more chance that a local byte variable will be globally assigned to a register. However, if the variable is used as an array or string subscript, then better code will be generated if it is a halfword.

FIXED(15) Versus FIXED(16)

In general, if a particular halfword variable is not required to be signed, then it is better to declare it is as FIXED(16) rather than FIXED(15) (which is the default). This is particularly true when halfwords are to be added to PTR or FIXED(32) fields. On the 8100, generated code could be larger if the halfword is FIXED(15):

```
*DCL P   PTR;
*DCL H16 FIXED(16);
*DCL H15 FIXED(15);
*P=P+H16;                          /* UNSIGNED ADDITION */
         AHR   P,H16
         AYHRE P,0                 /* (total 4 bytes)   */
*P=P+H15;                          /* SIGNED ADDITION   */
         AHR   P,H15
         AYHRE P,0
         AHRI  H15,0
         JNM   @SG00001
         SYHRE P,0                 /* (total 10 bytes)  */
@SG00001 DS    0H
```

When a halfword is added to a pointer as part of a subscript calculation, the shorter sequence is used whether the subscript is signed or unsigned.

On the other hand, if FIXED(16) fields are compared, then in some instances two branch instructions will be required whereas only one would be required with signed halfwords:

```
*DCL (H16,I16)  FIXED(16);
*DCL (H15,I15)  FIXED(15);
*IF H16 >= I16 THEN              /* LOGICAL COMPARISON */
         CHR   I16,H16
         JE    @RT00005
         JY    @RF00005
@RT00005 DS    0H
         - - -
         - - -
*IF H15 >= I15 THEN              /* ARITHMETIC COMPARISON */
         CHR   H15,I15
         JL    @RF00007
         - - -
         - - -
```

Use Structures for Overlay Defining Local Variables

A local variable can be given more than one data type by the construction of a suitable structure. This is preferable to the use of the DEFINED attribute which inhibits optimization. In this example:

```
DCL 1 XF16    FIXED(16),
      2 XF15    FIXED(15),
        3 XCH2    CHAR(2),
          4 XB16    BIT(16);
```

...the structure would allow the data item to be manipulated as any one of four data types. It may be declared as a register or may be assigned to a register by the optimization phase of the compiler.

Name Individual Bits

Flag bits in structures should be named, set, and tested individually. This has the following benefits:

- The code is more readable and easier to change. Use of masks makes code obscure and difficult to modify.
- The compiler will optimize references to bits in the same byte or halfword, as described next, under, "Group BIT Assignments...".

For example:

```
DCL 1 B BIT(8),
      2 B1  BIT(1),
      2 B23 BIT(2),
       - - -
      2 B8  BIT(1);
B1='1'B;
B23='01'B;
```

...is better than:

```
DCL B BIT(8);
B=B | 'C0'X;
B=B & 'BF'X;
```

...and is just as efficient.

Group Together BIT Assignments and their Tests

If consecutive BIT assignment statements set bits in the same byte or halfword they will be combined by the compiler. They will <u>not</u> be combined unless the statements are consecutive.

In the following example, swapping statements 5 and 6 would make three consecutive BIT assignments, and allow all three BIT assignments to be combined into one machine instruction. Note that the compiler has assigned byte F to a register.

```
2       *DCL 1 F BIT(8),
        *        3 FLAG1 BIT(1),
        *        3 FLAG2 BIT(1),
        *        3 FLAG3 BIT(1);
3       *FLAG1='1'B;
4       *FLAG2='1'B;
                 ORI   FLAG2,X'C0'
5       *X=0;
                 XHR   X,X
6       *FLAG3='1'B;
                 ORI   FLAG3,X'20'
```

Similar guidelines should be followed when coding IF statements. For example, coding:

```
IF FLAG1='1'B&
   X=0 &
   FLAG2='1'B THEN...
```

...would not allow the two "FLAG" tests to be combined. Better code would be obtained in this example if the last two tests were transposed.

Make Local Flags BIT(8) Structures

Local flags should be declared in BIT(8) structures. For example:

```
DCL 1 LCLFLAGS BIT(8),
      2 OPTSW  BIT(1),
    - - -
      2 ERRSW  BIT(1);

DCL YES BIT(1) CONSTANT('1'B);
DCL NO  BIT(1) CONSTANT('0'B);

LCLFLAGS=''B;           /* INITIALIZE LOCAL SWITCHES */
- - -
OPTSW=YES;              /* SET                       */
- - -
IF ERRSW=NO THEN        /* TEST                      */
```

The compiler will attempt to globally assign such structures to registers. The flags can be zeroed by LCLFLAGS=''B and efficient code will be generated for setting and testing individual flags.

Use of AUTODATA(n)

The AUTODATA procedure option will cause a diagnostic to be generated if the amount of AUTOMATIC data generated exceeds the specified value. If either NOAUTODATA or AUTODATA(0) is specified then it is clear that the user would (if possible) like all local variables assigned to registers, recalculating expressions if necessary. The compiler global register assignment will inspect this option and give preference to local variables, if necessary.

Sometimes the compiler is unable to assign enough data to registers to satisfy this user constraint on AUTODATA size. This will result in a message, IRE2909I E AUTODATA LIMIT OF nn BYTES HAS BEEN EXCEEDED BY YY BYTES...

In this case, you may have to study the assembly phase listing to determine why the excessive autodata occurred.

Avoid Using MACGEN

The use of MACGEN may reduce the effects of global optimization. There are options that can be specified on the MACGEN statement to tell the compiler which variables are referenced, for example. This information should always be specified. Failure to do this could cause either incorrect code to be generated or optimization to be reduced.

Code SIZE Option on MACGEN

The PL/DS Compiler does "goto optimization," that is, it generates short jumps rather than long jumps whenever the target of the jump is in range of the short jump. However, if a MACGEN statement is specified, the compiler may not know the size of the generated code within the MACGEN. The SIZE option enables the user to inform the compiler what this size is. If it is not specified, a default value of SIZE(20) is assumed and long jumps may be generated when short jumps would have sufficed (or the opposite, which may lead to an assembly failure).

Use Different Names For "Different" Variables

If the variable "I", for example, is used as a work variable in many parts of the program, then if it is globally assigned to a register it will be in the same register throughout the program. If different names (e.g. 'I1', 'I2', etc.) are used when the variable is used for essentially different functions, then better register assignment may be possible, since different registers can be used in different parts of the program. Better documentation may be a by-product of such a change.

The generated code will never degrade as a result of such a change.

Preventing Dead Code Elimination

If an unconditional branch is followed by code that is seemingly inaccessible, (i.e. does not have a label and is not a MACGEN) the compiler will, under certain circumstances, delete this section of code. Prefixing the section of code with a label will ensure that this does not happen.

"Jump-To-Jump" Optimization

The compiler has jump-to-jump optimization, which can reduce the size of branching code. If a branch is outside the range of a short jump the compiler will attempt to find an intermediate jump that already jumps to the desired target or, failing this, will insert one at a suitable point (i.e. after an unconditional branch). If there is no suitable point, then code for a long branch will be generated, which will be costly. In this case, it may be worth creating a suitable point for the compiler to make use of.

Example:

```
          J(NEXT):   /* THIS IS A SUITABLE POINT AFTER WHICH
                        AN INTERMEDIATE JUMP CAN BE INSERTED */
     NEXT: ;
```

Note: Do not code GOTO NEXT, since the compiler may "optimize it out." The use of a built-in instruction ensures that it does not get deleted.

SPECIALIZED CODING TECHNIQUES

This section contains hints that, if followed, could lead to an improvement in generated code but that are not generally recommended since they reduce code readability.

Local Copies of Basing Expressions

Use this hint with care, as described above.

If a routine makes many references to items in a control block that has a complex basing expression, the expression may have to be re-evaluated a number of times within the program. This may be for one of the following reasons:

1. If the base item in the basing chain is a declared register, the expression will be re-evaluated after a CALL since that register could have been modified.

2. There may not have been a register free to hold the expression.

3. It may have been more profitable to use the available registers for other expressions.

If it is known that the address of the control block does not change over a section of code, then its address can be assigned to a local pointer once and then the structure RESPECIFIED to be based on the local pointer. If this local pointer is then globally assigned to a register, the generated code will be considerably improved.

For example:

```
DCL 1 S1 BASED,
      2 S1A,
      2 S1B,
      2 S12P PTR,        /* -> S2 */
       - - -
       - - -
      2 S1Z;
RFY S1 BASED(S1P);
DCL 1 S2 BASED,
      2 S2A,
       - - -
       - - -
      2 S2Z;
RFY S2 BASED(S12P);
```

If many references are made to S2 items, then it may be worth adding these statements at the start of the module:

```
DCL LS2P     PTR;   /*          LOCAL PTR TO S2                  */

LS2P=ADDR(S2);      /* GET ADDRESS OF S2 INTO LOCAL POINTER */

RFY S2 BASED(LS2P); /* NOW USE LS2P TO BASE S2                  */
```

Local Copies of EXTERNAL/BASED Variables

Use this hint with care, as described above.

Wherever an EXTERNAL or BASED variable is set in a program then code will be generated to store it at that point. If a particular such variable is set many times in a program then it can sometimes be worth taking a local copy and doing one final assignment at the return point of the of the module. Note that code will only be improved if there is a register in which to hold the local copy of the variable.

Use of Pointers Instead of Subscripts/Substrings

Use this hint with care, as described above.

The use of arrays in PL/DS does carry a certain overhead in generated code. In instances where very efficient code is required, it may be worth using BASED variables to simulate the array addressing code. This is particularly true if iterative DO-loops are used.

Note: The source code may become less readable with the use of pointers rather than arrays.

For example:

```
DCL A(10)    FIXED(8);
DCL MAXVAL   FIXED(8);
DCL I        FIXED(8);
MAXVAL=0;
DO I=1 TO DIM(A);
   MAXVAL=MAX(MAXVAL,A(I));
END;
```

...could be coded:

```
DCL A(10)    FIXED(8);
DCL AB       FIXED(8) BASED(AP);
DCL AP       PTR;
DCL MAXVAL   FIXED(8);
DCL I        FIXED(8);
MAXVAL=0;
AP=ADDR(A);
DO I=1 TO DIM(A);
   MAXVAL=MAX(MAXVAL,AB);
   AP=AP+LENGTH(A);
END;
```

The WORKREGS Procedure Option

Use this hint with care, as described above.

The WORKREGS procedure option allows the user to specify which registers will be free for use as work registers during code generation. The option need not usually be specified since in its absence the compiler will choose work registers according to statement type and complexity.

Use of the WORKREGS option can sometimes improve generated code by increasing the number of registers available for global assignment. At the same time, however, the likelihood of running out of registers during code generation will be greater. Hence this option should be used with caution.

For example:

```
T1:PROC OPTIONS(REENTRANT,WORKREGS(4,6));
```

...specifies that halfword/word registers 4 and 6 are work registers and will not be available for global assignment.

NOWORKREGS specifies that no registers are to be reserved for code generation and therefore all registers can be used for global assignment (where they are not restricted).

For example:

```
T2:PROC OPTIONS(REENTRANT,NOWORKREGS);
```

Programs that make heavy use of REGISTER variables or where many registers are user-restricted are most likely to benefit from the use of the WORKREGS option. One approach may be to try compiling with NOWORKREGS, if the compilation fails with "Insufficient Registers for Code Generation" try WORKREGS(4) and so on.

CODING COMMON FUNCTIONS

This section demonstrates ways of coding common functions in PL/DS. The suggested code includes machine instructions, if necessary, to get the desired result.

Testing For Power of 2

For the following example:

```
DCL A   FIXED(16);
```

...this pseudo code gives the logic for finding if A is a power of 2:

```
IF A ¬= 0 &
   (A & (A-1))=0 THEN
  A is a power of 2.
ELSE
  A is not a power of 2.
```

Note: This will not work for negative values.

Testing for Odd/Even Values

For the following example:

```
DCL X FIXED(16);
```

...this pseudo code gives the logic to test whether X is odd or even:

```
IF X//2=0 THEN
  'X' is even
ELSE
  'X' is odd
```

Note: This can be generalized for testing for multiples of any constant:

```
IF X//16=0 THEN
  'X' is multiple of 16
```

Rounding Down

In the following example:

```
DCL X FIXED(16);
X=X/8*8;        /* ROUND DOWN TO NEAREST MULTIPLE OF 8 */
```

...the value of X is rounded down to the nearest multiple of 8.

Rounding Up

In the following example:

```
DCL X FIXED(16);
X=(X+7)/8*8;    /* ROUND UP TO NEAREST MULTIPLE OF 8 */
```

...the value of X is rounded up to the nearest multiple of 8.

Handling Strings Longer Than 256 Bytes

Here are two examples of moving strings longer than 256 bytes.

Example 1:

```
DCL (S,T) CHAR(4000) BDY(HWORD);
S=T;                 /* INVALID SINCE > 256 BYTES */
DCL C     CHAR(256)  BDY(HWORD) BASED;
DCL P     PTR;       /* TARGET POINTER            */
DCL Q     PTR;       /* SOURCE POINTER            */
DCL L     FIXED(15);
P=ADDR(S);           /* INITIALIZE TARGET POINTER */
Q=ADDR(T);           /* INITIALIZE SOURCE POINTER */
L=LENGTH(S);         /* INITIALIZE LENGTH TO MOVE */
DO WHILE(L > 256);
  P->C=Q->C;
  L=L-LENGTH(C);     /* DECREMENT LENGTH TO MOVE  */
  P=P+LENGTH(C);     /* INCREMENT TARGET POINTER  */
  Q=Q+LENGTH(C);     /* INCREMENT SOURCE POINTER  */
END;
IF L > 0 THEN        /* IF STILL BYTES TO MOVE    */
 P->C(1:L)=Q->C(1:L); /* MOVE REMAINDER           */
```

Example 2: Using MVHS Built-In Instruction.

```
DCL (S,T) CHAR(4000)   BDY(HWORD);
DCL (P,Q) PTR:
DCL H     FIXED(16);   /* NUMBER OF HALFWORDS TO MOVE */
DCL TEMPH FIXED(16);   /* TEMP # HWORDS FOR EACH MOVE */
P=ADDR(S);             /* INITIALIZE TARGET POINTER   */
Q=ADDR(T);             /* INITIALIZE SOURCE POINTER   */
H=LENGTH(S)/2;         /* INITIALIZE # HWORDS TO MOVE */
DO UNTIL(H <=0);
  TEMPH=H;             /* GET # HWORDS FOR THIS MOVE  */
  MVHS(P,Q,TEMPH);     /* MOVE 'B' HALFWORDS          */
  H=H-256;             /* DECREMENT # HWORDS TO MOVE  */
END;
```

Zeroing/Blanking Strings, Structures, and Arrays

The null bit string constant, written ''B, and the null character string constant, written '', often provide the best means of setting an area to zeroes or blanks. The language rule is that padding is with zeroes if either the source or target is BIT.

Example 1: String or structure.

```
DCL S CHAR(20);
S=''B;                  /* SETS S TO ZEROES */
S='';                   /* SETS S TO BLANKS */
```

Example 2: Array.

```
DCL TAB(20) CHAR(4);
DCL 1 TABC DEF(TAB),
      3 *(DIM(TAB)) CHAR(LENGTH(TAB));
TABC='';        /* SETS TAB TO BLANKS */
```

Example 3: Variable length string.

The statement

```
S(1:N)=''B
```

...is invalid since a variable length target cannot be longer than the source. A valid way of clearing a variable length string is as follows:

```
DCL S CHAR(20);
S(1:N)=S(1:N) && S(1:N);      /* SETS S(1:N) TO ZEROES */
```

See "Propagation of Characters," below, for setting S(1:N) to blanks.

Propagation of Characters

Sometimes you want to propagate a character in a variable without simply coding a literal value, because the number of character positions is too large.

Example 1: Set every column of CARD CHAR(80) to '*'.

```
DCL CARD  CHAR(80);
DCL P     PTR;
DCL BCARD CHAR(LENGTH(CARD)-1) BASED(P);
P=ADDR(CARD);
BCARD(1)='*';
EVAL(P+1)->BCARD=BCARD;
```

Note: CARD(2:80)=CARD(1:79) will not propagate the first character since the compiler recognizes that the source and receiver overlap and code is generated to place the contents of columns 1-79 into columns 2-80. When BASED character fields are used, overlap is only detected if the basing expressions are identical.

Example 2: Setting the first N characters of CARD CHAR(80) to blank.

```
DCL CARD  CHAR(80);
DCL P     PTR;
DCL BCARD CHAR(80) BASED(P);
P=ADDR(CARD);
BCARD(1)=' ';
EVAL(P+1)->BCARD(2:N-1)=BCARD(1:N-1);
```

Note: CARD(1:N)=''; will not work since a variable substring receiver is not permitted to be longer than the source.

Locating a Specified Character in a Character String

After a CLS instruction has been executed the address registers will be pointing at the first characters that did not compare. The following example shows how CLS can be used to find the first non-blank character in CARD:

```
DCL CARD  CHAR(80);
DCL (P,Q) PTR;
DCL I     FIXED(15);
P=ADDR(CARD);              /* POINT TO CARD                    */
Q=ADDR(' ');               /* POINT TO A BLANK                 */
I=1;                       /* SET LENGTH FOR FIRST COMPARE     */
CLS(Q,P,I);                /* TEST FIRST CHAR OF CARD          */
JNE(FOUND);                /* BRANCH IF FIRST CHAR BLANK       */
  DO;                      /* FIRST CHAR IS NOT BLANK          */
    Q=ADDR(CARD);          /* POINT TO FIRST CHAR (BLANK)      */
    I=LENGTH(CARD)-1;      /* SET LENGTH FOR SECOND COMPARE    */
    CLS(Q,P,I);            /* DO OVERLAPPING COMPARE           */
    JE(ALLBLANK);          /* ALL CHARACTERS ARE BLANK         */
  END;
FOUND:P=P-1;               /* NON-BLANK FOUND, ADJUST ADDRESS  */

/* P NOW ADDRESSES FIRST NON-BLANK CHARACTER IN CARD */
```

The above sequence can be improved by arranging for CARD to be preceded in storage by a blank.

Using a Branch Table

Example 1:

```
DCL LTABLE(3) PTR INIT(
        ADDR(LAB1),
        ADDR(LAB2),
        ADDR(LAB3));       /* LAB1 ETC. LABELS IN THIS PROC  */
DCL BLABEL LABEL BASED VALRG(LAB1,LAB2,LAB2);
                           /* VALRG IS REQD IN PL/DS         */
GOTO LTABLE(I)->BLABEL;    /* BRANCH TO I'TH LABEL IN LTABLE */
```

Example 2:

```
DCL ENT1  ENTRY EXTERNAL;
DCL ENT2  ENTRY EXTERNAL;
DCL ENT3  ENTRY EXTERNAL;
DCL ETABLE(3) PTR INIT(
        ADDR(ENT1),
        ADDR(ENT2),
        ADDR(ENT3));
DCL BENTRY ENTRY BASED VALRG(*);

CALL ETABLE(I)->BENTRY;    /* CALL I'TH ENTRY IN ETABLE */
```

Manipulation of Variable Bit Positions

The following example shows how individual bit positions in variables can be addressed and set in PL/DS:

```
DCL B       BIT(8);        /* FIELD TO BE SUBSTRINGED          */
DCL BITMASK1(8) BIT(8)
    INIT('80'X,'40'X,'20'X,'10'X,'08'X,'04'X,'02'X,'01'X);
DCL BITMASK2(8) BIT(8)
    INIT('7F'X,'BF'X,'DF'X,'EF'X,'F7'X,'FB'X,'FD'X,'FE'X);
B=B | BITMASK1(N);         /* SET N'TH BIT OF B ON             */
B=B & BITMASK2(N);         /* SET N'TH BIT OF B OFF            */
IF  (B & BITMASK1(N)) ¬= '00'X THEN /* TEST N'TH BIT OF B       */
  N'th bit of B is on.
ELSE
  N'th bit of B is off.
```

Manipulation of Bytes Within a Halfword

The best way to handle individual bytes within a halfword is to declare the halfword as a structure:

```
DCL 1 H FIXED(16),
      3 HK FIXED(8),
      3 HL FIXED(8);

/* HK AND HL CAN NOW BE MANIPULATED SEPARATELY */
/* H MAY NOW BE ASSIGNED TO THE LOWER HALFWORD */
/* PORTION OF A REGISTER                       */
```

Manipulation of Upper Halfwords of Registers

Upper halfwords in registers can be handled explicitly.

```
DCL 1 P PTR REG(6),
      2 PU FIXED(16),
      2 PL FIXED(16);
PU=0;
            - - -
            - - -
PU=PU+1;
            - - -
            - - -
IF PU=X THEN...
```

Use of Local @INCLUDEs (@CREATE)

The following example shows how the use of local includes together with the listing segmentation facility can improve code readability:

```
@PROCESS SOURCE(SEGMENT);
    @CREATE SYSUT5(AAA);
      DO;
          - - -
          - - -
      END
    @ENDCREATE;
    @CREATE SYSUT5(BBB);
      DO;
          - - -
          - - -
      END;
    @ENDCREATE;
@EJECT;
T:PROC;
  - - -
  /* THE USE OF INCLUDES ALLOWS THE LOGIC FOR THE MAIN  */
  /* LINE OF THIS ROUTINE TO BE SPECIFIED ON ONE PAGE.  */
  /* HOWEVER, IT IS NOT NECESSARY TO UPDATE A MACLIB    */
  /* TO MAKE CHANGES TO THE INCLUDES.                   */
  IF ERRFLAG=YES THEN
    @INCLUDE SYSUT5(AAA);
  ELSE
    @INCLUDE SYSUT5(BBB);
  - - -
  - - -
END;
```

Concatenating Character Strings

The following example shows how the MVS and STNI instructions can be used to perform concatenation:

```
DCL OUTREC CHAR(81);
DCL F1     CHAR(40);
DCL F2     CHAR(40);
DCL TP     PTR;          /* HOLD TARGET ADDRESS */
DCL SP     PTR;          /* SOURCE ADDRESS      */
DCL L      FIXED(15);

/* THE FOLLOWING IS EQUIVALENT TO OUTREC=F1 || '/' || F2; */

TP=ADDR(OUTREC);
SP=ADDR(F1);
L=LENGTH(F1);
MVS(TP,SP,L);            /* MOVE F1 TO OUTREC */
STNI('/',TP);            /* APPEND '/'        */
SP=ADDR(F2);
L=LENGTH(F2);
MVS(TP,SP,L);            /* APPEND F2         */
```

Zero-Origin Arrays

The following example shows how zero-origin arrays can be simulated in PL/DS:

```
DCL A(10) INIT((10)0);
DCL 1 * BASED(ADDR(AB)),
      2 A0   FIXED(16),
      2 AZ(DIM(A)-1) FIXED(16);
X=A(I);  /* I CAN RANGE FROM 1-10 */
X=AZ(I); /* I CAN RANGE FROM 0-9  */
X=A0;    /* SPECIAL-CASE SINCE AZ(0) WOULD BE INVALID */
```

Coding an Infinite Loop

The following will cause an infinite loop to be generated:

```
DO UNTIL(1=2);
END;
```

Determining Register Restrictedness

The register restrict status is included as a 32-bit mask (8 hex characters) in the assembly phase listing of each generated instruction. A one-bit in the nth position of the mask means that the nth register is one of the following:

1. Restricted by the user at that instruction.

2. Restricted by the compiler at the instruction since the register contains a value needed at a later instruction.

3. It is a work register temporarily restricted during code generation for the one statement.

To determine whether a register is user-restricted at a particular statement:

1. Compile with the NOOPTIMIZE option.

2. Add a labeled null statement at the required point in the program.

The mask against this statement will have bits on for user-restricted and globally restricted (e.g. SAVEREG) registers only. For example, consider the following sample line from the listing produced by the assembly phase:

```
                                    (restrict mask) (statement number)
                                          |              |
                                          V              V
LAB          DS     0H                  300003C0 0005
```

Here, the mask 300003C0 has the bit value:

```
00110000000000000000001111000000.
```

The "1" bits indicate that bytes 2, 3, 22, 23, 24, and 25 (or lower halfwords 2, 22 and 24) in registers are restricted at statement 5.

Avoiding Corruption of a Register by an EXTERNAL Procedure

It is possible, if necessary, to restrict a register before the external PROCEDURE by coding a RESPECIFY statement in front of it:

```
RFY (RH18,RH20) RSTD;
E:PROC OPTIONS(...
  @INCLUDE SYSLIB(REGDCLS);
  - - -
  - - -
```

Restricting a register before an external PROCEDURE statement may occasionally be necessary to prevent corruption of these registers in the prologue code.

Using a Translate Table

Example:

```
        DCL TRTAB(256) CHAR(1) INIT(
             '20'X,'21'X,.....
              - - -
             'CF'X);
        DCL TRANSLTE(256) CHAR(1) BASED(ADDR(TRTAB)) POS(2);
        DCL (T,C) CHAR(1);
        DCL CARD CHAR(80);

   /* SINGLE CHARACTER */
        T=TRANSLTE(C);

   /* STRING */
        DO I=1 TO LENGTH(CARD);
          CARD(I)=TRANSLTE(CARD(I));
        END;
```

Use of the CTLZ Machine Instruction

The CTLZ (Count Leading Zeroes) instruction can be very useful in certain circumstances. Remember that it returns in the first operand the number of leading zeros contained in the second operand. It also turns off the first one bit in the second operand. For example:

```
        DCL N       FIXED(16);
        DCL H       BIT(16);

        H='0001010011110000'B;
        CTLZ(N,H);
   /* N is 3;                    */
   /* H is '0000010011110000'B;  */
```

CTLZ can also be used to test a halfword for a power of two and find which power it is:

```
        DCL N       FIXED(16);
        DCL H       FIXED(16);
        DCL TH      FIXED(16);
        TH=H;
        IF TH ¬=0 THEN
          DO;
            CTLZ(N,TH);
            GNZ(NOTP2);   /* BRANCH IF SOME BITS STILL ON */
            N=15-N;
            POWER2:       /* H WAS 2**N                   */

          END;
        NOTP2:            /* H WAS NOT A POWER OF 2       */
```

Binary to Character Conversion

The following PL/DS procedure converts a halfword binary field into EBCDIC, making use of the DHR instruction.

```
BINCHAR:
    PROC(BINFLD,CHARFLD);
         /*****************************************************************/
         /* INPUT PARAMETER DECLARATIONS                                  */
         /*****************************************************************/
         DCL BINFLD   FIXED(16),                 /* BINARY INPUT          */
             CHARFLD  CHAR(Z#DIGITS);            /* CHARACTER OUTPUT      */
         /*****************************************************************/
         /* LOCAL DECLARATIONS                                            */
         /*****************************************************************/
         DCL Z#DIGITS FIXED CONSTANT(5);     /* NO. OF OUTPUT DIGITS      */
         /*****************************************************************/
         /* THE FOLLOWING TWO REGISTERS MUST FORM A 'HALFWORD PAIR'       */
         /* FOR USE IN THE 'DHR' INSTRUCTION.                             */
         /*****************************************************************/
         DCL 1 REM    FIXED(16) REG(4) UNRSTD,   /* REMAINDER             */
               2 *      FIXED(8);
               2 LREM   FIXED(8);                /* LOW ORDER BYTE OF REM */
         DCL LBINFLD FIXED(16) REG(6) UNRSTD;   /* LOCAL COPY OF INPUT */
         DCL I       FIXED(8);                  /* LOOP CONTROL VARIABLE */
         /*****************************************************************/
         /* TAKE WORK COPY OF INPUT PARAMETER                             */
         /*****************************************************************/
         RFY (REM,LBINFLD) RSTD;
         LBINFLD=BINFLD;                        /* TAKE LOCAL COPY       */
         /*****************************************************************/
         /* DO OVER OUTPUT FIELD                                          */
         /*****************************************************************/
         DO I=Z#DIGITS BY -1 TO 1;              /* DO OVER OUTPUT FIELD  */
           REM=0;                               /* INITIALIZE REMAINDER  */
           DHR(REM,10);                         /* DIVIDE BY 10 AND GET
                                                   REMAINDER             */
           CHARFLD(I)='F0'X+LREM;               /* WRITE OUT THIS DIGIT  */
         END;
    END;                              /* BINCHAR                         */
```

Character to Binary Conversion

The following PL/DS procedure converts an EBCDIC character field into binary.

```
CHARBIN:
     PROC(CHARFLD,BINFLD);
         /*****************************************************************/
         /* INPUT PARAMETER DECLARATIONS                                  */
         /*****************************************************************/
         DCL CHARFLD  CHAR(Z#DIGITS),            /* CHARACTER OUTPUT      */
             BINFLD   FIXED(16);                 /* BINARY INPUT          */
         /*****************************************************************/
         /* LOCAL DECLARATIONS                                            */
         /*****************************************************************/
         DCL Z#DIGITS FIXED CONSTANT(5);     /* NO. OF OUTPUT DIGITS      */
         DCL I        FIXED(8);              /* LOOP CONTROL VARIABLE     */
         BINFLD=0;                           /* INITIALIZE                */
         /*****************************************************************/
         /* DO OVER INPUT FIELD                                           */
         /*****************************************************************/
         DO I=1 TO Z#DIGITS;                 /* DO OVER OUTPUT FIELD      */
           BINFLD=BINFLD*10+(CHARFLD(I)&'0F'X); /* ACCUMULATE RESULT      */
         END;
    END;                              /* CHARBIN                         */
```

SOURCE CODE PARAMETERIZATION

It is good practice to code in such a way that specification changes can be made with minimum changes to the source code. The PL/DS language makes parameterization very easy with the CONSTANT attribute and built-in functions such as LENGTH, DIM, MAX, and MIN. Following is just one example showing how new entries can be added to the look-up table (LTAB) or the function table (FTAB) without any further changes being necessary:

```
        DCL C      CHAR(1);              /* CHAR TO LOOK UP    */
        DCL 1 LTAB STATIC,               /* LOOKUP TABLE       */
              2 LTABEL,
                3 * CHAR(1)   INIT('B'),
                3 * FIXED(8)  INIT(1),
              2 *,
                3 * CHAR(1)   INIT('D'),
                3 * FIXED(8)  INIT(1),
              2 *,
                3 * CHAR(1)   INIT('E'),
                3 * FIXED(8)  INIT(2),
              2 *,
                3 * CHAR(1)   INIT('H'),
                3 * FIXED(8)  INIT(3);
        DCL 1 LOOKUP(LENGTH(LTAB)/LENGTH(LTABEL))
                             BASED(ADDR(LTAB)),
              3 INCHAR   CHAR(1),
              3 OUTINDEX FIXED(8);
        DCL 1 FTAB STATIC,               /* FUNCTION ROUTINE ADDRESSES */
              3 FTABEL PTR INIT(ADDR(FUNC1)),
              3 *      PTR INIT(ADDR(FUNC2)),
              3 *      PTR INIT(ADDR(FUNC3));
        DCL FUNC(LENGTH(FTAB)/LENGTH(FTABEL)) PTR DEF(FTAB);
        DCL ROUTINE ENTRY BASED(FUNC(OUTINDEX(I))) VALRG(*);
/*************************/
/* SEARCH TABLE FOR MATCH */
/*************************/
      DO I=1 TO DIM(LOOKUP)
                  WHILE(C¬=INCHAR(I));
      END;
/*****************************/
/* BRANCH TO FUNCTION ROUTINE */
/*****************************/
      IF I > DIM(LOOKUP) THEN
        CALL ERR;
      ELSE
        CALL ROUTINE;
```

APPENDIX H. PL/DS OPTIONS AND CONTROLS

This appendix is a quick reference for users of this publication. For complete and definitive descriptions of options and controls, consult the publication: Programming Language for Distributed Systems (PL/DS), User's Guide.

COMPILER OPTIONS

ADECK (NOADECK) Compiler Option

Function: Causes the module produced by the assembly phase to be placed on the SYSPUNCH data set. (NOADECK: Suppresses ADECK option).

Parameters: None.

Default: NOADECK.

ADEFS (NOADEFS) Compiler Option

Function: Causes all variables, referenced and unreferenced, to be defined in the assembler code. (NOADEFS: define only referenced variables.)

Parameters: None.

Default: NOADEFS.

ALIST (NOALIST) Compiler Option

Function: Causes assembly listing to be printed (NOALIST: no listing.)

Parameters: None.

Default: ALIST.

ANNOTATE (NOANNOTATE), ANNO (NOANNO) Compiler Option

Function: Causes PL/DS source statements to appear as comments in the assembler text, preceding the assembler code that is generated for each statement.

Parameters: None.

Default: ANNOTATE.

ASSEMBLE (NOASSEMBLE), ASM (NOASM) Compiler Option

Function: Allows the final assembly phase of the compiler to be invoked if no errors more severe than the specified threshold (I, W, E, or S) were encountered in the compile phase. (NOASSEMBLE: suppresses ASSEMBLE option.)

Parameters: (I) (W) (E) (S).

Default: ASSEMBLE (E)

ATITLE (NOATITLE) Compiler Option

Function: Allows specification of identifier to appear in assembly listing titles and in SYSLIN (see OBJECT compiler option) object module records.

Parameter: ('name'). A name of up to four characters.

Default: NOATITLE

AXREF (NOAXREF) Compiler Option

Function: Causes on the assembly listing a cross reference of symbols used in the assembly phase. (NOAXREF: no cross reference.)

Parameters: None.

Default: AXREF.

BUFSIZE, BUF Compiler Option

Function: Indicates how much contiguous storage for INCLUDE buffers is needed after storage is allocated for the SIZE option.

Parameters: (bytesK), (bytes).

Default: BUFSIZE(20K).

COMPILE (NOCOMPILE), COMP (NOCOMP),C (NOC) Compiler Option

Function: Causes the compile phase to be executed.

Parameters: None.

Default: COMPILE.

ESD (NOESD) Compiler Option

Function: Causes printing of the external symbol dictionary (ESD) as part of the listing. The options ASSEMBLE and ALIST must be in effect. (NOESD: no ESD printed.)

Parameters: None.

Default: ESD.

EXTEND Compiler Option

Function: Allows OPTIMIZE to be used on large programs by increasing the space available for statically defined tables used by them.

Parameters: (nK).

Default: EXTEND(0).

FDECK (NOFDECK), FD (NOFD) Compiler Option

Function: Causes the writing of formatted source statements to the SYSPUNCH data set. The option FORMAT must be in effect. (NOFDECK: no formatted SYSPUNCH.)

Parameters: None.

Default NOFDECK.

FLAG, F Compiler Option

Function: Controls level of diagnostic messages printed: Information, Warning, Error, Serious.

Parameters: (I) (W) (E) (S).

Default: FLAG(I).

FORMAT (NOFORMAT), FMT (NOFMT) Compiler Option

Function: Invokes the formatter, which produces a listing with comments aligned, and with logical alignment of nested IF, THEN, ELSE, DO, and END structured programming sequences. Causes either the macro source or the compiler source to be formatted. You can submit both FORMAT(COMPILE) and FORMAT(MACRO) in the same run.

Parameters: None, or (MACRO) (COMPILE) (COMP).

Default: If FORMAT without operands, FMT(COMP); if not specified, NOFORMAT.

FSOURCE (NOFSOURCE), FS (NOFS) Compiler Option

Function: Causes the source before formatting to be included in the compiler information listing.

Parameters: None.

Default: NOFSOURCE.

IDR (NOIDR) Compiler Option

Function: In the compile phase, causes a CSECT Information Data Record to be added to the END statement of each external procedure.

Parameters: None.

Default: IDR.

LINECOUNT, LC Compiler Option

Function: Defines number of lines, including heading, printed on each page.

Parameter: (number of lines).

Default: LINECOUNT(60).

MACPARM, MPARM Compiler Option

Function: Causes a string of parameter data to be passed to macro processing phase, where it is accessed by the MACPARM macro definition function.

Parameter: ('string').

Default: MPARM('').

MACRO (NOMACRO), M (NOM) Compiler Option

Function: Causes macro statement processing by the macro phase of the compiler. (NOMACRO: no macro processing phase.)

Parameters: None.

Default: MACRO

MARGINS, MAR Compiler Option

Function: Defines the first and last positions of control and source statement records to be read by the compiler.

Parameters: (left column, right column).

Default: MARGINS(2,72).

MDECK (NOMDECK), MD (NOMD) Compiler Option

Function: Causes the writing of the source statements, as modified by the macro processing phase of the compiler, to the SYSPUNCH data set.

Parameters: None.

Default: NOMDECK.

MPERCENT, MPER Compiler Option

Function: Causes the specified percentage of the SIZE option to be used for the global dictionary and string area.

Parameter: (Value).

Default: MPERCENT(20).

MSOURCE (NOMSOURCE), MS (NOMS) Compiler Option

Function: Causes the printing of the PL/DS source statements before they are modified by the macro phase of the compiler.

Parameters: None.

Default: NOMSOURCE.

MXREF (NOMXREF) Compiler Option

Function: Causes the macro variable attribute and cross-reference to be printed.

Parameters: None.

Default: If MSOURCE, then MXREF; if NOMSOURCE, then NOMXREF.

OBJECT (NOOBJECT) Compiler Option

Function: With DPPX parameter, places the object module in the SYSITEXT data set for 8100 object modules (lrecl=256). With DPDS parameter, places text data in the SYSLIN data set for simulator object modules (lrecl=80). With BOTH parameter, does both DPPX and DPDS actions.

Parameters: DPDS, DPPX, BOTH.

Default: OBJECT (DPDS).

OPTIMIZE (NOOPTIMIZE), OPT (NOOPT) Compiler Option

Function: Indicates that the compiler should optimize the code for execution speed (TIME) or for size of compiled module(SPACE). (NOOPTIMIZE: minimal optimization of code.)

Parameters: SPACE or TIME or none.

Default: If OPTIMIZE specified with no parameters, OPTIMIZE(SPACE). Otherwise, NOOPTIMIZE.

OPTIONS (NOOPTIONS), OPTN (NOOPTN) Compiler Option

Function: Causes printing, at the beginning of the listing, of the compiler options used.

Parameters: None.

Default: OPTIONS.

RLD (NORLD) Compiler Option

Function: Causes printing of the relocation dictionary in the listing. The options ASSEMBLE and ALIST must be in effect.

Parameters: None.

Default: RLD.

SIZE, SZ Compiler Option

Function: Allows specification of the size of the dynamic area for the internal compiler dictionary and text buffers. Minimum dynamic area for dictionary is 32K (32,768) bytes.

Parameters: (bytesK), (bytes), (MAX).

Default: SIZE(MAX). (See User's Guide)

SOURCE (NOSOURCE), S (NOS) Compiler Option

Function: Causes PL/DS source statements to be printed in segmented or non-segmented form. (The SEGMENT or NOSEGMENT parameter of the %INCLUDE or @INCLUDE statement can override the SEGMENT/NOSEGMENT parameter of the SOURCE option.) If macro processing took place, the modified source is printed. (See MSOURCE compiler option for programmer dataset listing.)

Parameters: None, (SEGMENT), (SEG), (NOSEGMENT), (NOSEG).

Default: SOURCE(NOSEGMENT).

STATISTICS (NOSTATISTICS), STATS (NOSTATS)

Function: Produces a list of statisics at end of compiler run.

Parameters: None

Default: STATISTICS

TERMINAL (NOTERMINAL), TERM (NOTERM) Compiler Option

Function: Causes error messages from the compile phase to be written to SYSTERM. (NOTERMINAL: no diagnostic messages written to SYSTERM.)

Parameters: None.

Default: NOTERM.

TEST (NOTEST) Compiler Option

Function: Causes special source symbol table (SYM cards) in object module.

Parameters: None.

Default: TEST.

TITLE Compiler Option

Function: Allows specification of a heading for each page of listing.

Parameter: ('title'). Maximum of 63 characters.

Default: First 63 characters of first non-PROCESS statement.

XREF (NOXREF), X (NOX) Compiler Option

Function: Causes attribute and cross reference table to be printed in the listing.

Parameters: None.

Default: XREF if SOURCE; NOXREF if NOSOURCE.

There can be no other statement except a comment on a compiler control statement line. The @ sign prefix must be the first non-blank character after the left margin of the first line of a compiler control statement. Each statement can take as many additional lines as required to complete.

@CREATE, @ENDCREATE -- Define Program Segments for Later Inclusion

Purpose

To facilitate structured programming, defines program segments to the compiler, for later inclusion with an @INCLUDE or %INCLUDE statement.

Rules

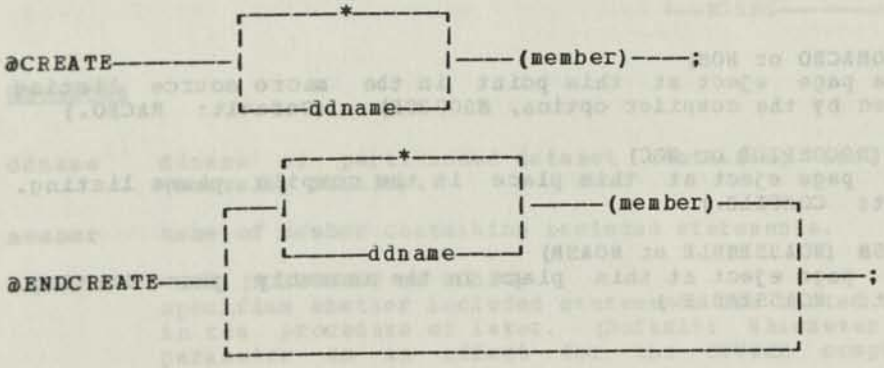
Must follow any @PROCESS cards.

Any number of segments may be defined, each segment enclosed by an @CREATE/@ENDCREATE pair.

Defined segments may not contain nested @CREATE and @ENDCREATE commands.

The "at" sign (@) prefix must be the first character, within margins, in the line. The @CREATE or @ENDCREATE must be the only statement, other than a comment, in the line.

Syntax



Operands

ddname	name of PDS where member is to reside. (Normally SYSUT5.)
*	ddname is interpreted as SYSUT5.
member	name of member holding this segment.

@EJECT -- Start New Page in Listing

Purpose

Cause the compiler, under conditions specified in the options, on this statement, to begin a new listing page on the next source statement. (See also @SPACE.)

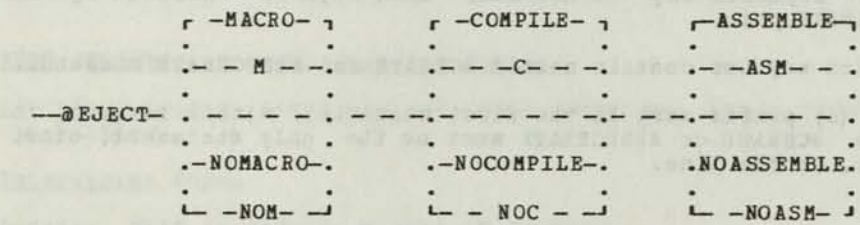
Rules

Options may be specified in any order.

Code anywhere in source code.

The "at" sign (@) prefix must be the first character, within margins, of the line. The @EJECT must be the only statement, other than a comment, in the line.

Syntax



Operands

MACRO or M (NOMACRO or NOM)
force a page eject at this point in the macro source listing (produced by the compiler option, MSOURCE). (Default: MACRO.)

COMPILE or C (NOCOMPILE or NOC)
force a page eject at this place in the compile phase listing. (Default: COMPILE.)

ASSEMBLE or ASM (NOASSEMBLE or NOASM)
force a page eject at this place in the assembly phase listing. (Default: NOASSEMBLE.)

@INCLUDE -- Include Data Sets Containing Source Statements

Purpose

Incorporate contents of specified partitioned data set member, containing compiler source statements, into this run at this point. The data set member can be one created with @CREATE/@ENDCREATE at the beginning of this compile.

Rules

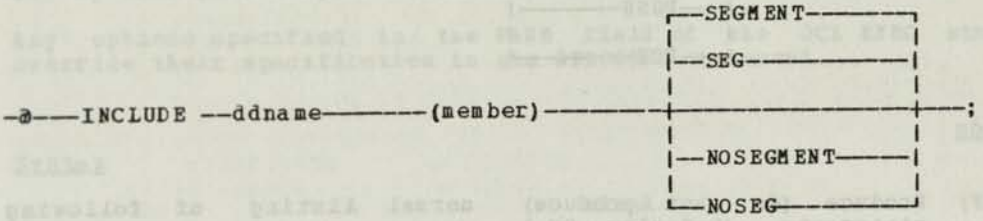
The same statement may be used more than once.

Each statement must be on one line only.

The "at" sign (@) prefix must be the first character, within margins, in the line. The @INCLUDE must be the only statement, other than a comment, in the line.

In contrast to the macro %INCLUDE statement, the @INCLUDE is not performed by the compiler until after the macro processing phase.

Syntax



Operands

ddname — ddname of partitioned dataset containing included member. (Normally SYSLIB).

member — name of member containing included statements.

SEGMENT or SEG (NOSEGMENT or NOSEG)
specifies whether included statements are listed at this point in the procedure or later. (Default: Whichever segmentation parameter is in effect for the SOURCE compiler control option.)

@LIST -- Control Contents of Listing

Purpose

Controls whether or not the part of the program following this statement is to be printed on the compiler listing. Or saves present LIST ON or LIST OFF status.

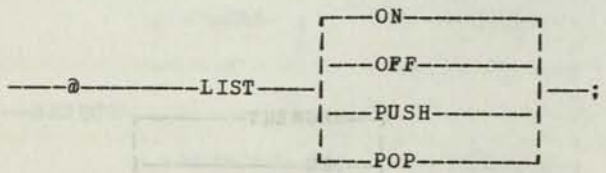
Rules

Remains in effect until next @LIST command.

Each statement must be on one line only.

The "at" sign (@) prefix must be the first character, within margins, in the line. The @LIST must be the only statement, other than a comment, in the line.

Syntax



Operands

ON (OFF) Produce (do not produce) normal listing of following statements. (Default: ON.)

PUSH Save the present (but not necessarily known) ON or OFF LIST status for later resumption using the POP option.

POP Resume the LIST OFF or LIST ON status saved with the most recent LIST PUSH statement.

@PROCESS -- Specify Compiler Options for This Compile

Purpose

Allows you to specify any compiler options that are not already specified in the PARM field of the compiler EXEC statement.

Rules

The PROCESS control statement(s) must appear before your source statements and other control statements.

All characters in the @PROCESS statement must be within the margins specified in the MARGINS compiler option (default 2,72).

Must precede the other source code and control statements.

The @PROCESS can be the only statement, other than a comment, in the record.

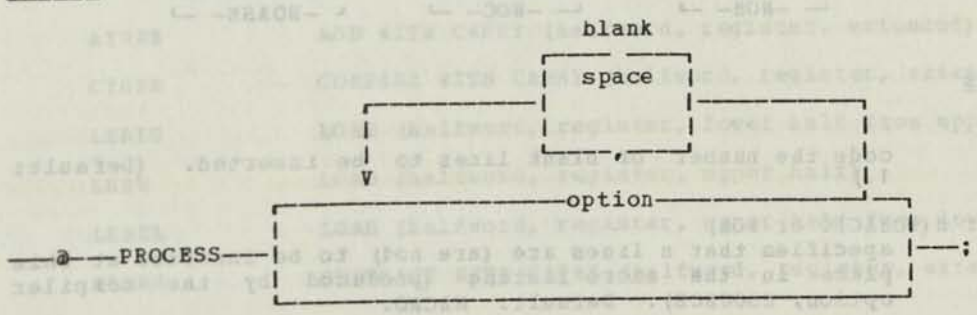
The "at" sign (@) prefix must be the first character, within margins, in the record.

May have continuation lines, with @PROCESS on the initial line only.

The options in the option field may be in any sequence.

Any options specified in the PARM field of the JCL EXEC statement override their specification in the @PROCESS statement.

Syntax



@SPACE -- Leave Blank Lines on Listing

Purpose

Cause the compiler, under conditions specified in the options on this statement, to insert this number of spaces before printing the next line.

Rules

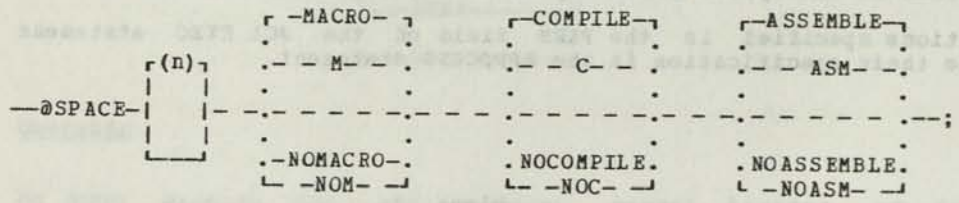
You may insert an @SPACE anywhere in the source text dataset.

You may code the options in any sequence after the number.

The compiler stops inserting blank lines at the bottom of the current page of the listing or when the specified number of lines has been inserted, whichever comes first.

The "at" sign (@) prefix must be the first character, within margins, in the record. The @SPACE must be the only statement, other than a comment, in the record.

Syntax



Operands

n
: code the number of blank lines to be inserted. (Default: 1.)

MACRO or M(NOMACRO or NOM)
: specifies that n lines are (are not) to be inserted at this place in the macro listing (produced by the compiler option, MSOURCE). Default: MACRO.

COMPILE or C (NOCOMPILE or NOCOMP)
: specifies that n lines are (are not) to be inserted at this place in the compiler source listing (produced under the compiler option, SOURCE). Default: COMPILE.

ASSEMBLE or ASM (NOASSEMBLE OR NOASM)
: specifies that n lines are to be inserted at this place in the assembly listing (produced under the compiler option, ASOURCE). Default: NOASSEMBLE.

APPENDIX I. SUMMARY OF PL/DS REGISTER NOMENCLATURE

The publication, 8100 Information System: Principles of Operation has a complete description of the program registers on the 8100.

For programmer reference, the following is a summary of the conventions used by the DPPX Assembler to view the registers. It is important to know the register conventions in order to use them in PL/DS statements, machine instructions, and assembly code. Refer to Figure 39 as you read these descriptions.

THE PRIMARY AND SECONDARY REGISTERS

The 8100 has sixteen word registers. The word registers are even-numbered: 0, 2, 4, 6 ... 30. The first eight word registers, numbered 0 through 14, are the primary word registers.

The second eight word registers, numbered 16 through 30, are the secondary word registers.

Each primary and secondary word register is subdivided into:

- an upper halfword
- a lower halfword consisting of two bytes.

UPPER HALFWORD OF REGISTERS

The upper halfword of each primary and secondary word register is explicitly addressable with the following machine instruction mnemonics:

AYHRE	ADD WITH CARRY (halfword, register, extended)
CYHRE	COMPARE WITH CARRY (halfword, register, extended)
LHRLU	LOAD (halfword, register, lower half from upper)
LHRU	LOAD (halfword, register, upper half)
LHRUL	LOAD (halfword, register, upper half from lower)
SYHRE	SUBTRACT WITH CARRY (halfword, register, extended)

The upper halfword of each register is indirectly addressable with high level compile code by declaring a structure, as shown in the "Example of Register Addressability," below.

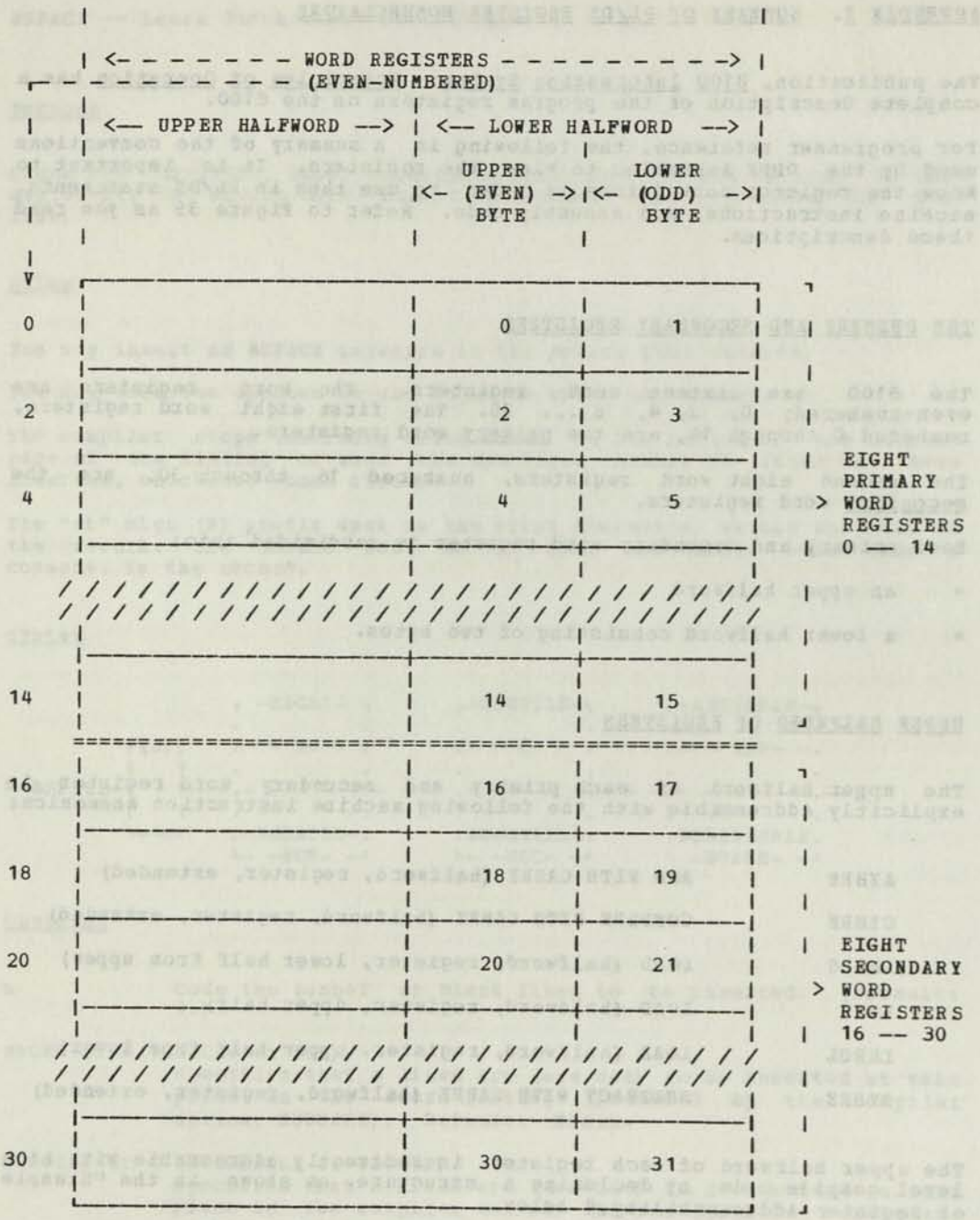


Figure 39. DPPX Assembler Register Conventions

LOWER HALFWORD OF REGISTERS

The lower halfword portion of each primary and secondary word register is an even-odd numbered pair of bytes.

The halfword is addressable and so are the bytes; that is the upper and lower byte of each lower halfword.

The upper byte of the pair has the same number as the word register it is in, and the lower byte has that number plus one. For example, word register 6 contains bytes 6 and 7 in its lower halfword.

EXAMPLE OF REGISTER ADDRESSABILITY WITH PL/DS CODE

The following is an example of using high level PL/DS compile code to establish addressability within a register, including the upper halfword, by declaring a register structure.

```
DECLARE 1 WR    FIXED(32)  REGISTER 8,
          2 WRUH  FIXED(16),        /* REG 8 UPPER HALFWORD      */
          2 WRLH  FIXED(16),        /* REG 8 LOWER HALFWORD      */
            3 WRHB  CHAR(1),        /* REG 8 HIGH ORDER BYTE (8) */
            3 WRLB  CHAR(1);        /* REG 8 LOW ORDER BYTE (9)  */
```

USING THE REGISTERS

When an implied or explicit byte in a register is required, PL/DS uses one of the byte positions that to PL/DS's knowledge is not in use.

When an implied or explicit halfword in a register is required, PL/DS uses one of the lower halfwords that, to PL/DS's knowledge, does not have one of its bytes already in use.

When an implied or explicit word in a register is required, PL/DS uses one of its word registers that, to PL/DS's knowledge, has neither its lower nor upper halfword in use.

Take note of the expression "to PL/DS's knowledge," above. When you as a programmer submit code at the register level, you are responsible for restricting, with the RESPECIFY command, each register that you want reserved for use in your code.

You should use the RESPECIFY command to free up (UNRESTRICT) your restricted registers at the point in your code where your program is finished with them.

EXTENT OF REGISTER RESTRICTION WITH RESPECIFY

When you use the RESPECIFY statement for register restriction, the amount of reserved register space depends upon the data type attribute in your original declaration of the specified register variable.

For example:

```
DECLARE R2F FIXED(32) REGISTER(2);

RESPECIFY R2F RESTRICTED;
```

...restricts word register 2.

Also:

```
DECLARE R2H FIXED(15) REGISTER(2);

RESPECIFY R2H RESTRICTED;
```

...restricts the lower halfword of register 2, and gives the compiler the right to use the upper halfword of register 2.

Further:

```
DECLARE R2B FIXED(8) REGISTER(2);

RESPECIFY R2B RESTRICTED;
```

...restricts byte 2 (the upper byte of the lower halfword) of word register 2, and gives the compiler the right to use the low-order byte

of the lower halfword of word register 2 and the upper halfword of word register 2.

Similarly:

```
DECLARE R3B FIXED(8) REGISTER(3);

RESPECIFY R3B RESTRICTED;
```

...restricts the low-order byte of the lower halfword of word register 2, and gives the compiler the right to use the high-order byte of the lower halfword of word register 2 and the upper halfword of word register 2.

T

U

Distributed Processing Development System (DPDS)
Programming Language for Distributed Systems (PL/DS)
Reference

READER'S COMMENT FORM

Order No. SC27-0446-0

This manual is part of a library that serves as a reference source for systems analysts, programmers, and operators of IBM systems. This form may be used to communicate your views about this publication. They will be sent to the author's department for whatever review and action, if any, is deemed appropriate. Comments may be written in your own language; use of English is not required.

IBM may use or distribute any of the information you supply in any way it believes appropriate without incurring any obligation whatever. You may, of course, continue to use the information you supply.

Note: *Copies of IBM publications are not stocked at the location to which this form is addressed. Please direct any requests for copies of publications, or for assistance in using your IBM system, to your IBM representative or to the IBM branch office serving your locality.*

How did you use this publication?

[ ] As an introduction
[ ] As a reference manual
[ ] For another purpose (explain) _____

[ ] As a text (student)
[ ] As a text (instructor)

_____

Is there anything you especially like or dislike about the organization, presentation, or writing in this manual? Helpful comments include general usefulness of the book; possible additions, deletions, and clarifications; specific errors and ommissions.

Page Number: **Comment:**

Note: Staples can cause problems with automated mail sorting equipment. Please use pressure sensitive or other gummed tape to seal this form.

What is your occupation? _____

Newsletter number of latest Technical Newsletter (if any) concerning this publication: _____

If you wish a reply, give your name and address:

IBM branch office serving you _____

Thank you for your cooperation. No postage stamp necessary if mailed in the U.S.A. (Elsewhere, an IBM office or representative will be happy to forward your comments or you may mail directly to the address in the Edition Notice on the back of the title page.)

SC27-0446-0

Reader's Comment Form

Cut or Fold Along Line

Fold and tape | Please Do Not Staple | Fold and tape

NO POSTAGE NECESSARY IF MAILED IN THE UNITED STATES

BUSINESS REPLY MAIL

FIRST CLASS PERMIT NO. 40 ARMONK, NEW YORK

POSTAGE WILL BE PAID BY ADDRESSEE

International Business Machines Corporation
Department 63T
Neighborhood Road
Kingston, New York 12401

Fold and tape | Please Do Not Staple | Fold and tape

International Business Machines Corporation
Data Processing Division
1133 Westchester Avenue, White Plains, N.Y. 10604

IBM World Trade Americas/Far East Corporation
Town of Mount Pleasant, Route 9, North Tarrytown, N.Y., U.S.A. 10591

IBM World Trade Europe/Middle East/Africa Corporation
360 Hamilton Avenue, White Plains, N.Y., U.S.A. 10601

DPDS Programming Language for Distributed Systems (PL/DS) Reference (File No. 8100/S370-31) Printed in U.S.A. SC27-0446-0

Distributed Processing Development System (DPDS)
Programming Language for Distributed Systems (PL/DS)
Reference

Order No. SC27-0446-0

READER'S COMMENT FORM

This manual is part of a library that serves as a reference source for systems analysts, programmers, and operators of IBM systems. This form may be used to communicate your views about this publication. They will be sent to the author's department for whatever review and action, if any, is deemed appropriate. Comments may be written in your own language; use of English is not required.

IBM may use or distribute any of the information you supply in any way it believes appropriate without incurring any obligation whatever. You may, of course, continue to use the information you supply.

Note: *Copies of IBM publications are not stocked at the location to which this form is addressed. Please direct any requests for copies of publications, or for assistance in using your IBM system, to your IBM representative or to the IBM branch office serving your locality.*

How did you use this publication?

[ ] As an introduction
[ ] As a reference manual
[ ] For another purpose (explain) _____

[ ] As a text (student)
[ ] As a text (instructor)

_____

Is there anything you especially like or dislike about the organization, presentation, or writing in this manual? Helpful comments include general usefulness of the book; possible additions, deletions, and clarifications; specific errors and ommissions.

Page Number: Comment:

What is your occupation? _____

Newsletter number of latest Technical Newsletter (if any) concerning this publication: _____

If you wish a reply, give your name and address:

IBM branch office serving you _____

Thank you for your cooperation. No postage stamp necessary if mailed in the U.S.A. (Elsewhere, an IBM office or representative will be happy to forward your comments or you may mail directly to the address in the Edition Notice on the back of the title page.)

Note: Staples can cause problems with automated mail sorting equipment.
Please use pressure sensitive or other gummed tape to seal this form.

SC27-0446-0

Reader's Comment Form

Cut or Fold Along Line

DPDS Programming Language for Distributed Systems (PL/DS) Reference (File No. 8100/S370-31) Printed in U.S.A. SC27-0446-0

Fold and tape | **Please Do Not Staple** | Fold and tape

NO POSTAGE NECESSARY IF MAILED IN THE UNITED STATES

BUSINESS REPLY MAIL

FIRST CLASS **PERMIT NO. 40** **ARMONK, NEW YORK**

POSTAGE WILL BE PAID BY ADDRESSEE

International Business Machines Corporation
Department 63T
Neighborhood Road
Kingston, New York 12401

Fold and tape | **Please Do Not Staple** | Fold and tape

IBM®

International Business Machines Corporation
Data Processing Division
1133 Westchester Avenue, White Plains, N.Y. 10604

IBM World Trade Americas/Far East Corporation
Town of Mount Pleasant, Route 9, North Tarrytown, N.Y., U.S.A. 10591

IBM World Trade Europe/Middle East/Africa Corporation
360 Hamilton Avenue, White Plains, N.Y., U.S.A. 10601

Distributed Processing Development System (DPDS)
Programming Language for Distributed Systems (PL/DS)
Reference

READER'S COMMENT FORM

Order No. SC27-0446-0

This manual is part of a library that serves as a reference source for systems analysts, programmers, and operators of IBM systems. This form may be used to communicate your views about this publication. They will be sent to the author's department for whatever review and action, if any, is deemed appropriate. Comments may be written in your own language; use of English is not required.

IBM may use or distribute any of the information you supply in any way it believes appropriate without incurring any obligation whatever. You may, of course, continue to use the information you supply.

Note: *Copies of IBM publications are not stocked at the location to which this form is addressed. Please direct any requests for copies of publications, or for assistance in using your IBM system, to your IBM representative or to the IBM branch office serving your locality.*

How did you use this publication?

[ ] As an introduction
[ ] As a reference manual
[ ] For another purpose (explain) _____

[ ] As a text (student)
[ ] As a text (instructor)

_____

Is there anything you especially like or dislike about the organization, presentation, or writing in this manual? Helpful comments include general usefulness of the book; possible additions, deletions, and clarifications; specific errors and ommissions.

Page Number:	Comment:

What is your occupation? _____

Newsletter number of latest Technical Newsletter (if any) concerning this publication: _____

If you wish a reply, give your name and address:

IBM branch office serving you _____

Thank you for your cooperation. No postage stamp necessary if mailed in the U.S.A. (Elsewhere, an IBM office or representative will be happy to forward your comments or you may mail directly to the address in the Edition Notice on the back of the title page.)

Note: Staples can cause problems with automated mail sorting equipment.
Please use pressure sensitive or other gummed tape to seal this form.

SC27-0446-0

Reader's Comment Form

Cut or Fold Along Line

DPDS Programming Language for Distributed Systems (PL/DS) Reference (File No. 8100/S370-31) Printed in U.S.A. SC27-0446-0

Fold and tape | Please Do Not Staple | Fold and tape

NO POSTAGE NECESSARY IF MAILED IN THE UNITED STATES

BUSINESS REPLY MAIL

FIRST CLASS PERMIT NO. 40 ARMONK, NEW YORK

POSTAGE WILL BE PAID BY ADDRESSEE

International Business Machines Corporation
Department 63T
Neighborhood Road
Kingston, New York 12401

Fold and tape | Please Do Not Staple | Fold and tape

International Business Machines Corporation
Data Processing Division
1133 Westchester Avenue, White Plains, N.Y. 10604

IBM World Trade Americas/Far East Corporation
Town of Mount Pleasant, Route 9, North Tarrytown, N.Y., U.S.A. 10591

IBM World Trade Europe/Middle East/Africa Corporation
360 Hamilton Avenue, White Plains, N.Y., U.S.A. 10601

SC27-0446-0

DPDS Programming Language for Distributed Systems (PL/DS) Reference (File No. 8100/S370-31) Printed in U.S.A. SC27-0446-0

International Eusiness Machines Corporation
Data Processing Division
1133 Westchester Avenue, White Plains, N.Y. 10604

IBM World Trade Americas/Far East Corporation
Town of Mount Pleasant, Route 9, North Tarrytown, N.Y., U.S.A. 10591

IBM World Trade Europe/Middle East/Africa Corporation
360 Hamilton Avenue, White Plains, N.Y., U.S.A. 10601

Technical Newsletter

This Newsletter No.	SN31-1175
Date	June 15, 1980
Base Publication No.	SC27-0446-0
File No.	8100/S370/4300-31
Previous Newsletters	SN31-1093

Distributed Processing Development System
Programming Language for Distributed Systems
Reference

This Technical Newsletter (TNL) provides the following replacement pages for the subject publication:

Front Cover	73, 74
Title Page, Edition Notice	81, 82
iii, blank	82.1, blank
v–viii	85, 86
15, 16	86.1, 86.2
53-58	99, 100
58.1, 58.2	261–272
59, 60	Back Cover
68.1, 68.2	

A change to the text or to an illustration is indicated by a vertical line to the left of the change.

Summary of Amendments

This TNL reflects the addition of:

- Three compile statements: ITERATE, LEAVE, and SELECT
- Two pairs of procedure statement options: PATCHAREA/NOPATCHAREA and CODEBNDRY/NOCODEBNDRY

It also contains a rewritten section on looping do-groups and incorporates minor technical and editorial changes.

Note: *Please file this cover letter at the back of the manual to provide a record of changes.*

IBM Corporation, Department 63T, Kingston, New York 12401

Printed in U.S.A.

SC27-0446-0
File No. 8100/S370/4300-31

IBM 8100 Information System

Distributed Processing Development System Programming RPQ P88016

Programming Language for Distributed Systems Reference

Systems

Program Number: 5799-AZL

IBM

SC27-0446-0
File No. 8100/S370/4300-31

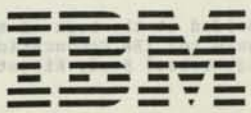
IBM 8100 Information System

Distributed Processing Development System Programming RPQ P88016

Programming Language for Distributed Systems Reference

Program Number: 5799-AZL

Systems



First Edition (May 1979)

This edition applies to Release 1, Modification Level 0, of the Distributed Processing Development System, program number 5799-AZL, an IBM PRPQ. The PRPQ described in this manual, and all licensed material available for it, are provided by IBM under terms of the Agreement for IBM Licensed Programs. Your branch office can advise you of ordering procedures.

Information in this manual is subject to change. Subsequent revisions or technical newsletters will include such changes.

Before using this publication in connection with the operation of IBM systems, consult your IBM representative to find out which editions are applicable and current.

It is possible that this material may contain reference to, or information about, IBM products (machines and programs), programming, or services that are not announced in your country. Such references or information must not be construed to mean that IBM intends to announce such IBM products, programming, or services in your country.

Copies of this and other IBM publications can be obtained through IBM branch offices.

A form for reader's comments has been provided at the back of this publication. Address any additional comments to: IBM Corporation, Product Publications, Department 63T, Neighborhood Road, Kingston, New York 12401.

IBM may use or distribute any of the information you supply in any way it believes appropriate without incurring any obligation whatsoever. You may, of course, continue to use the information you supply.

PREFACE

The Programming Language for Distributed Systems (PL/DS) Compiler and the Program Development Simulator on a System/370 or 4300 processor permit development and testing, in an interactive environment, of programs for the IBM 8100 Information System.

This manual provides information for coding source programs to submit to the PL/DS compiler. For the information to compile a program see the PL/DS User's Guide, in the list below.

The prerequisite publications are:

- Distributed Processing Development System General Information, GC27-0505
- IBM 8100 Information System: Principles of Operation, GA23-0031

The related manuals are:

- Distributed Processing Development System Programming Language for Distributed System User's Guide, SC27-0478
- Distributed Processing Development System Programming Language for Distributed Systems Macros for Distributed Processing Programming Executive Base Reference, SC27-0447
- Distributed Processing Development System Program Development Simulator User's Guide, SC27-0479
- Distributed Processing Programming Executive Assembler Programming: Language Reference and Guide
- Distributed Processing Programming Executive Base Programming: Guide to System Services

Use of This Publication

The parts of this manual that you will use depend upon your programming department's policies and standards, and your own assignment as a programmer. Beyond Chapter 1, which has basic information, you would use:

- Chapter 2, for high level PL/DS coding.
- Chapter 3, for coding machine language statements.
- Chapter 4, for modifying source code during a compile, and/or invoking macros.
- Chapter 5, for coding macro definitions.

For coding DPPX Assembler statements under the PL/DS compiler, see the description of MACGEN in Chapter 2.

Consult the Table of Contents for appendixes appropriate to your needs.

CONTENTS

FIGURES

MARGINS: Specify margins of input records that contain compiler source.

MDECK: Punch macro processing phase output to SYSPUNCH.

MPERCENT: Amount of SIZE option for global dictionary and string area.

MSOURCE: Print original source, before any macro processing, on listing.

MXREF: Print macro variable attribute and cross reference.

OBJECT: Format object code for simulator (SYSLIN), or for transfer to DPPX (SYSITEXT), or both.

OPTIMIZE: Optimize the code produced by the compile phase.

OPTIONS: Print compiler options at beginning of listing.

RLD: Print listing of the relocation dictionary.

SIZE: Specify dictionary and text buffer size.

SOURCE: Specify printing of source.

STATISTICS: Print compile phase statistics, such as number of statements.

TERMINAL: Specify message severity for SYSTERM.

TEST: Produce special source symbol table (SYM) object module records.

TITLE: Specify heading for each page of compile phase listing.

XREF: Print attribute and cross reference table on listing.

COMPILER CONTROL STATEMENTS

Besides the special PROCESS control statement described above for specifying compiler options, there are six other control statements.

As described above, under "Source Statement Format," there can be no other statement except a comment on a compiler control statement line. The @ sign prefix must be the first non-blank character after the left margin of the first line of a compiler control statement. Each statement can take as many additional lines as required to complete.

@CREATE: Begins and names a segment for later inclusion with an @INCLUDE or %INCLUDE statement.

@EJECT: Specifies the conditions for an eject to a new printout page at this place in the compiler-produced listing.

@ENDCREATE: Ends the segment started with the last @CREATE statement.

@INCLUDE: Incorporates compiler source statements from external libraries into this compilation.

@LIST: Allows changing, overriding, and restoring the control for printing compiler-generated lines after this place in the program. This applies to the listing after the compile phase.

@PROCESS: Specify compiler options. (Note: the PROCESS statement usage is described in the section, "Compiler Options.")

@SPACE: Leave blank line(s) on listing.

CHAPTER 2: COMPILE STATEMENTS AND BUILT-IN FUNCTIONS

This section describes the rules for coding compile statements and the built-in functions that the compiler supports for these statements.

In addition to these statements, you can code machine instructions, as described in the section, "Built-In Machine Instructions."

Besides in-line source code, you can bring the contents of source code data sets into your input.

Your set of source statements can be dynamically altered or added to during a compile run using the macro language. A macro phase of the compiler acts on your macro instructions and performs the specified alterations before the compile phase processes the final source statements. See the section, "Compiler Macro Language."

COMPILE STATEMENTS

SUMMARY OF COMPILE STATEMENTS

STATEMENT	USE
assignment	Dynamically assigns source expression value to variable.
CALL	Invokes an external or internal procedure.
DECLARE	Describes attributes of a variable.
DO	Starts a do-group.
ELSE	States action when IF condition is false.
END	Ends do-group or end of procedure.
ENTRY	Marks secondary entry points in a procedure.
MACGEN, ENDGEN	Identifies assembler text to be passed unchanged through compile phase.
GOTO	Transfers control (normally within procedure).
IF	States condition for IF, THEN, and (optionally) ELSE sequence.
ITERATE	Performs next iteration, or repetition, of a do-group.
LEAVE	Exits from a do-group.
null statement	Indicates no compiler action (fall through), usually in a THEN or ELSE.
PROCEDURE	Marks primary entry point in procedure.
RESPECIFY	Overrides attributes of variable.
RETURN	Returns control from a called procedure.
SELECT	Chooses execution path from a set of alternatives.
THEN	States action when IF condition is true.

DO -- Start of Do-Group

Purpose

The DO statement is a means of grouping statements into a common execution unit called a do-group. The DO statement begins the do-group; the END statement completes it. The statements within the do-group are treated as a single unit.

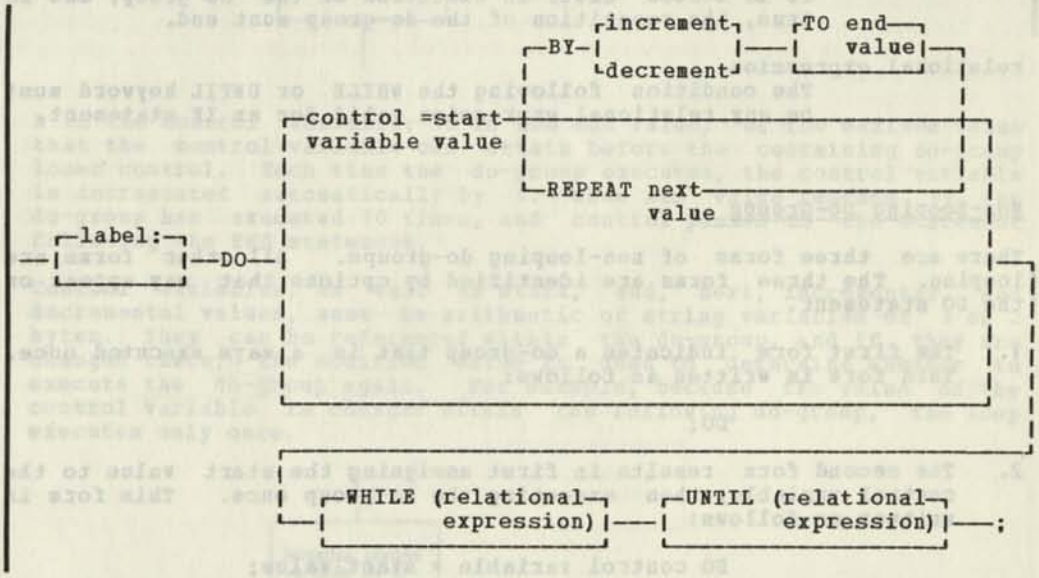
This single unit can be executed once, many times, or not at all. Non-looping do-groups are executed at most once; looping do-groups may be executed repeatedly.

Rules

The END statement marks the end of the do-group.

DO is a reserved keyword; do not use it as a variable name.

Syntax



Operands

label Optional. Code one or more labels, each followed by a colon.

control variable Name of variable to hold a value that controls the number of repetitions of the group. This must have attributes that are valid in an arithmetic context.

start value A constant, variable, or expression whose value is to be placed into the control variable.

BY Keyword that specifies that the control variable is to be bumped up or down at each iteration of the do-group.

increment Specifies the value to add to the control variable before comparing it to the end value.

decrement Specifies the value to subtract from the control variable before comparing it to the end value. If the first nonblank character after the keyword, BY, is a minus

sign, the BY value is a decrement instead of an increment.

TO — Keyword that specifies that an end value is to be compared to the control variable prior to each iteration of the do-group.

end value — A constant, variable, or expression that specifies the value of the control variable at which repetition of the do-group is to end.

REPEAT — Keyword that specifies that the value of the control variable is to be replaced by the associated expression value at each iteration of the do-group.

next value — A constant, variable, or expression that specifies the value to replace the value of the control variable at each iteration of the do-group.

WHILE — Keyword that specifies that the condition following it must be true for repetition of the do-group to occur.

UNTIL — Keyword that specifies that when the condition following it is tested after an execution of the do-group, and is true, the repetition of the do-group must end.

relational expression — The condition following the WHILE or UNTIL keyword must be any relational expression valid for an IF statement.

Non-Looping Do-Groups

There are three forms of non-looping do-groups. All other forms are looping. The three forms are identified by options that may appear on the DO statement.

1. The first form indicates a do-group that is always executed once. This form is written as follows:

```
DO;
```

2. The second form results in first assigning the start value to the control variable, then executing the do-group once. This form is written as follows:

```
DO control variable = start value;
```

3. The third form uses the WHILE condition to determine whether or not the do-group is executed. If the WHILE condition is met, this form results in one execution of the do-group. If the WHILE condition is not met, the do-group is not executed. This form is written as follows:

```
DO control variable = start value
   WHILE (  relational expression  );
```

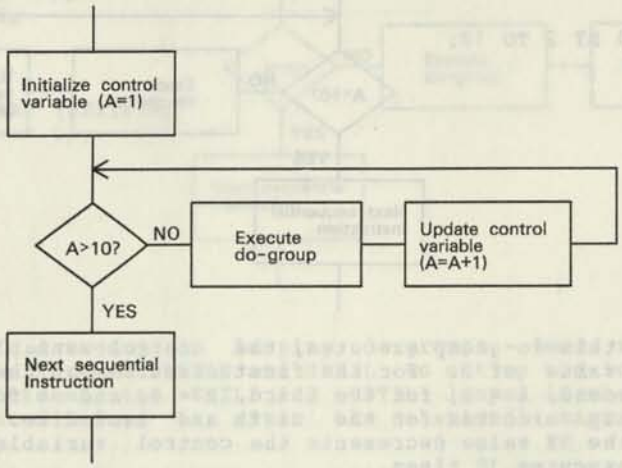
Looping Do-Groups

There are essentially two kinds of looping do-groups. In one, the compiler establishes a control variable, or counter, and modifies it each time a pass is made through the do-group. Setting up and modifying the control variable is accomplished through the use of constants, variables, arithmetic expressions, and combinations of the keywords BY, TO, and REPEAT. In the other kind of looping do-group, the compiler executes the do-group according to the validity of a relational expression after the keywords WHILE and UNTIL. Looping do-groups can have either or both forms of control. In the following examples,

flowcharts to the right of the PL/DS code illustrate the instruction flow in looping do-groups.

The following example illustrates the first kind of looping do-group. It executes 10 times before control passes to the statement following it.

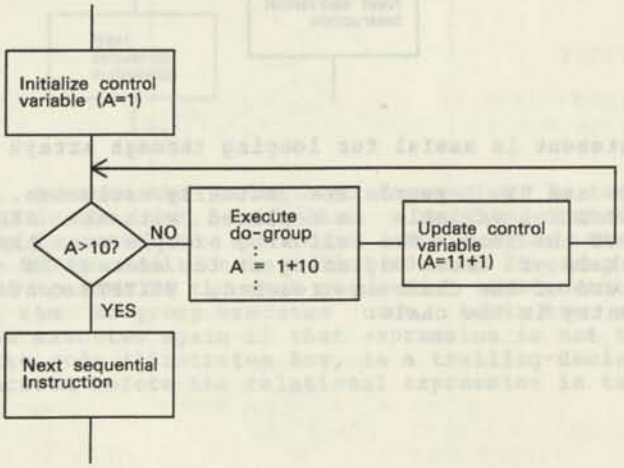
```
DO A = 1 TO 10;
   .
   .
   .
END;
```



A is the control variable; 10 is the end value, or the maximum value that the control variable can attain before the containing do-group loses control. Each time the do-group executes, the control variable is incremented automatically by 1. When its value reaches 11, the do-group has executed 10 times, and control passes to the statement following the END statement.

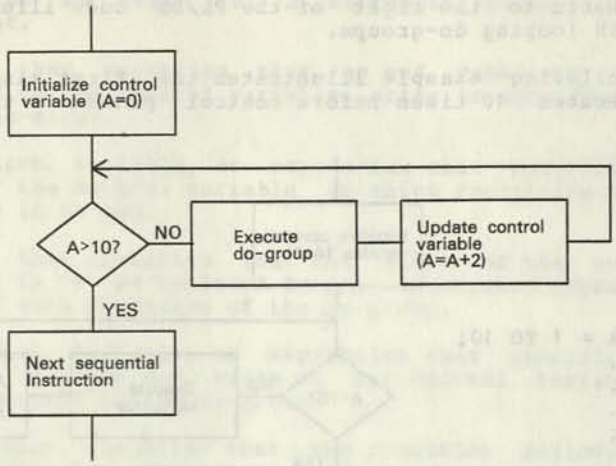
Control variables, as well as start, end, next, incremental, and decremental values, must be arithmetic or string variables of 1 or 2 bytes. They can be referenced within the do-group, and if they are changed there, the modified value is used to determine whether to execute the do-group again. For example, because the value of the control variable is changed within the following do-group, the loop executes only once.

```
DO A = 1 TO 10;
   .
   .
   A = A + 10;
   .
   .
END;
```



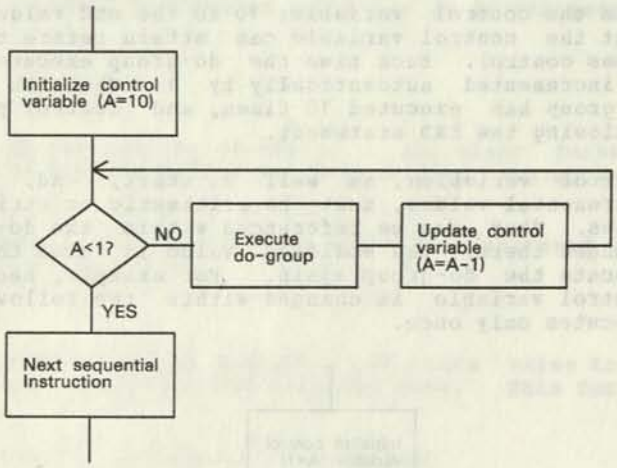
The next example illustrates how the counter or control variable can be incremented by the BY value. This do-group executes six times before control passes to the statement following it.

```
DO A = 0 BY 2 TO 10;
   .
   .
END;
```



Each time this do-group executes, the control variable A is incremented by the BY value of 2. For the first execution of the do-group, A = 0; for the second, A = 2; for the third, A = 4; and so forth. When A = 10, the do-group executes for the sixth and last time. In the following example, the BY value decrements the control variable, A, by 1, and the do-group executes 10 times.

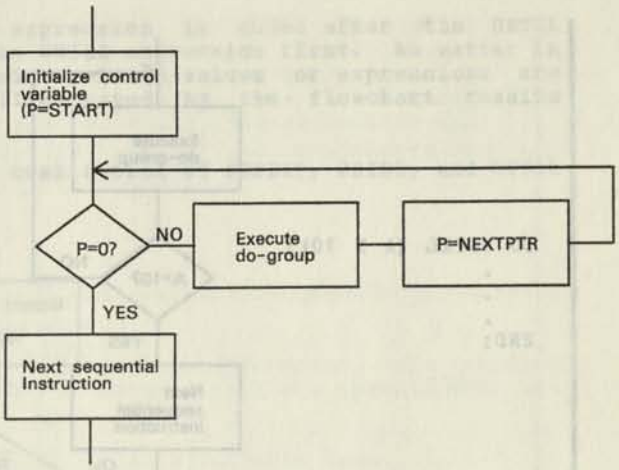
```
DO A = 10 BY -1 TO 1;
   .
   .
END;
```



The BY statement is useful for looping through arrays or strings.

The REPEAT and BY keywords are mutually exclusive. REPEAT indicates that the control variable is replaced with the REPEAT value at each execution of the loop. The following example uses the REPEAT keyword to search a chain of areas beginning at the address in the pointer START, until the end of the chain is reached. NEXTPTR contains the address of the next entry in the chain.

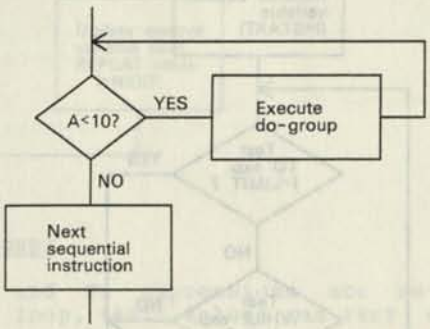
```
DCL (P, START) PTR;
DCL 1 BLOCK BASED (P),
    2 NEXTPTR PTR,
    .
    .     ;

DO P = START REPEAT (NEXTPTR)
       WHILE (P ¬ = 0);
   .
   .
END;
```



The second broad category of looping do-groups uses relational expressions with the keywords WHILE and UNTIL to set up the loops. The chief difference between the WHILE and UNTIL loops is whether the do-group is executed before or after the relational expression is tested.

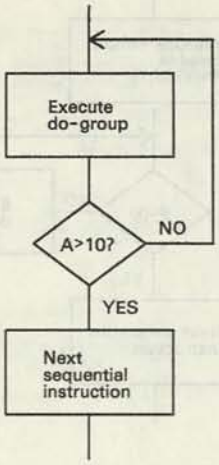
The DO-WHILE loop, because the relational expression is tested before each execution of the do-group, is called a leading-decision loop. If the WHILE condition is satisfied, the do-group is executed. In the following example, the do-group executes only if the relational expression, A < 10, is true. The flowchart next to the code illustrates how, in a leading-decision loop, the relational expression is tested before the do-group is executed.

```
DO WHILE (A < 10);
   .
   .
   .
END;
```



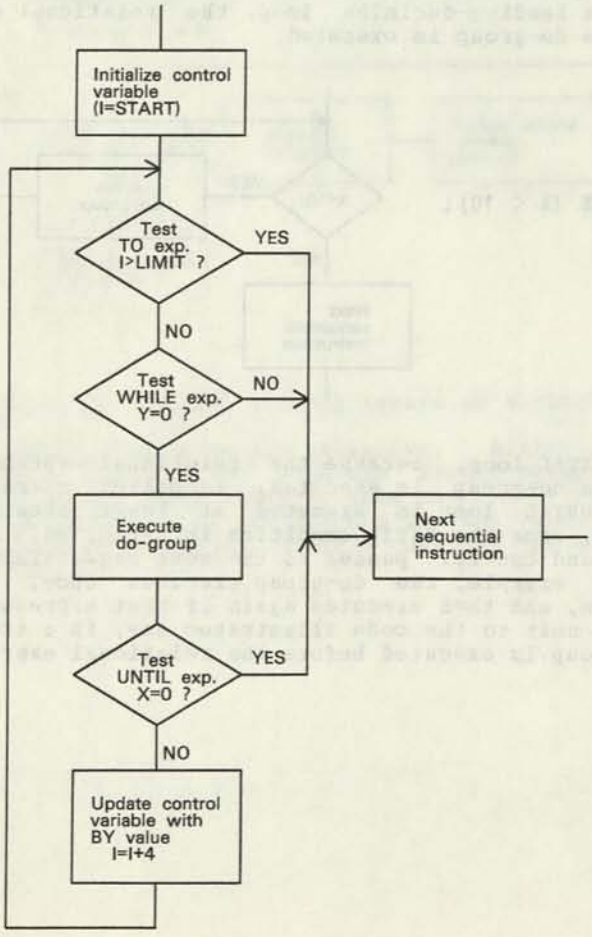
The DO-UNTIL loop, because the relational expression is tested only after the do-group is executed, is called a trailing-decision loop. Each DO-UNTIL loop is executed at least once. After the initial execution, once the UNTIL condition is satisfied, the do-group ceases to execute, and control passes to the next sequential instruction. In the following example, the do-group executes once, tests the relational expression, and then executes again if that expression is not true. The flowchart next to the code illustrates how, in a trailing-decision loop, the do-group is executed before the relational expression is tested.

```
DO UNTIL (A > 10);
   .
   .
   .
END;
```



BY and REPEAT may not be used in the same do-group, nor may TO and REPEAT. With the exception of those two pairs of mutually exclusive keywords, looping do-groups may combine keywords in any order. The following example illustrates the combination of BY, TO, WHILE, and UNTIL. The flowchart illustrates the sequence in which the compiler tests expressions and variables.

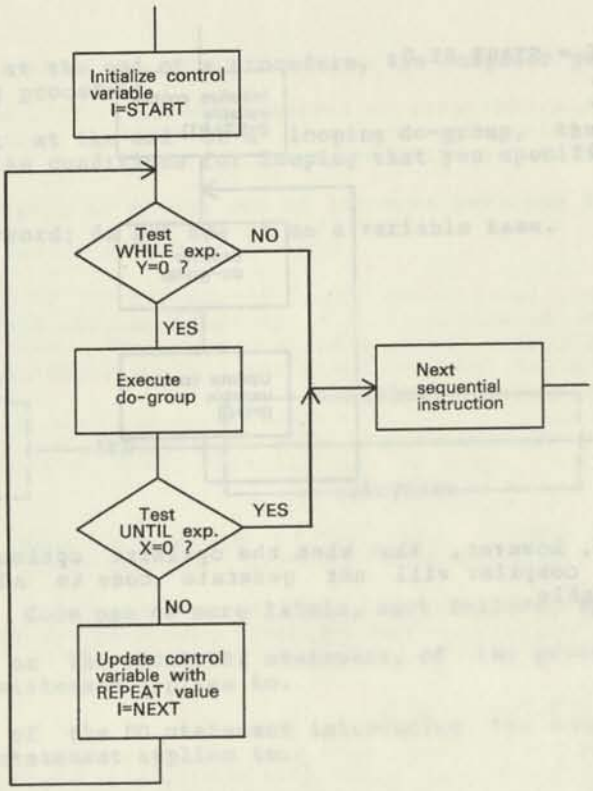
```
DO I = START
   BY 4
   TO LIMIT
   UNTIL (X = 0)
   WHILE (Y = 0);
   .
   .
   .
   .
END;
```



Note that although the WHILE expression is coded after the UNTIL expression, the compiler tests the WHILE expression first. No matter in what order the keywords and their related values or expressions are coded, the instruction flow illustrated by the flowchart remains unchanged.

The next example illustrates the combination of REPEAT, WHILE, and UNTIL keywords.

```
DO I = START
   REPEAT (NEXT)
   WHILE (Y = 0)
   UNTIL (X = 0);
   .
   .
   .
END;
```



Notes on Looping Do-Groups

1. Because the BY and TO expressions are re-evaluated on each iteration of the loop, their values can vary during the execution of the loop.

2. If the control variable and TO value have precision FIXED (8), an infinite loop results when the TO value reaches 255. If both values have precision FIXED (16), an infinite loop results when the TO value reaches 65,535.

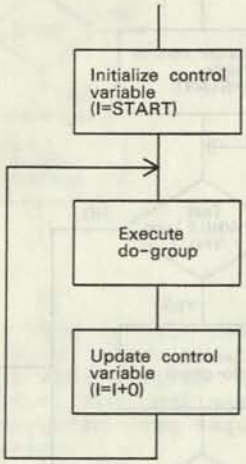
3. Loops can be represented without the use of many of the keywords associated with looping do-groups. The following example illustrates two ways of setting up a loop to perform the same task:

```
LOOP:  DO I = START
           BY DELTA
           WHILE (X = 0);
           .
           .
           .
       END;
```

```
LOOP:  I = START
REALHEAD:  IF X ¬ = 0 THEN
               GOTO EXIT;
           .
           .
       I = I + DELTA;
       GOTO REALHEAD;
EXIT:              ;
```

4. Infinitely looping do-groups may prove useful in some circumstances. A master control program, for example, might run on an infinitely looping do-group with a seldom-occuring execute command and exit nested deeply within it. The following example illustrates an infinitely looping do=group:

```
DO I = START BY 0;
   .
   .
   .
END;
```



Note, however, that when the optimize option is used to compile, the compiler will not generate code to add 0 to the control variable.

END -- End of Procedure or Do-Group

Purpose

Tells the compiler that this is the end of the statements for the present procedure, do-group, or select-group.

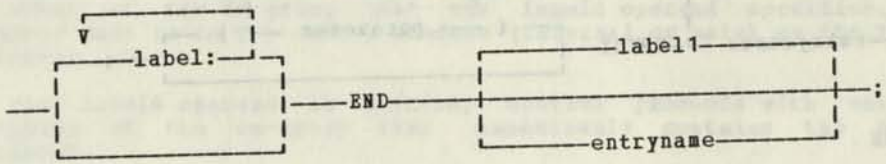
Rules

For an END statement at the end of a procedure, the compiler generates a return to the calling procedure.

For an END statement at the end of a looping do-group, the compiler generates a test of the conditions for looping that you specified in the DO statement.

END is a reserved keyword; do not use it as a variable name.

Syntax



Operands

label Optional. Code one or more labels, each followed by a colon.

entryname The name, on the PROCEDURE statement, of the procedure that this END statement applies to.

label1 The label of the DO statement introducing the do-group that this END statement applies to.

ENTRY -- Secondary Entry Point

Purpose

Designates a secondary entry point for a procedure. (The primary entry point is the procedure's PROCEDURE statement.)

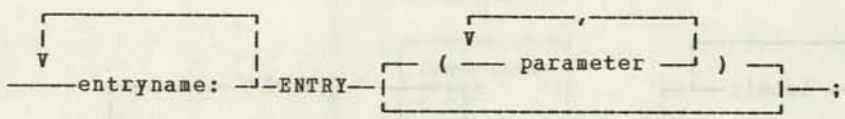
Rules

The number of secondary entry points in a procedure is not restricted.

Use the CALL statement to invoke a procedure at a secondary entry point. You may pass arguments with the CALL by coding their associated parameters on the ENTRY statement.

ENTRY is a reserved keyword; do not use it as a variable name.

Syntax



Operands

entryname Code at least one to specify the name you will use in the CALL statement for this secondary entry point.

parameter Code the names of the variables with which the procedure references the arguments passed in the CALL statement.

Declaring the Entry Name

When declaring the entry name that you reference in the ENTRY statement, the DECLARE statement appears in the calling procedure. If you do not declare the entry name, appropriate defaults are assigned when the entry name is encountered. For a detailed description of how to declare an entry name, refer to the discussion of "Program Data" in the DECLARE statement section.

Specifying Entry Arguments and Parameters

Parameters on an ENTRY statement must be simple data item names, each separated by a comma. The maximum number of parameters is 16. The number and position of the parameters on a PROCEDURE and ENTRY statement need not be the same.

There is a correspondence between the parameters on the ENTRY statement and the arguments on the CALL statement that invokes this entry point. When you include arguments on a CALL statement, a parameter list is produced. Each argument is assigned one word in the parameter list; an address is inserted in each word. The parmreg points to this parameter list across the linkage.

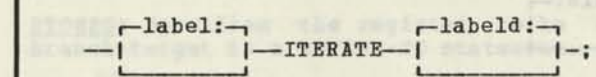
The compiler associates CALL statement arguments with ENTRY statement parameters in the order in which they appear; the first argument is associated with the first parameter, the second argument with the second

ITERATE -- Perform Next Iteration, or Repetition, of a Do-Group

Purpose

The ITERATE statement causes control to end the current iteration, or repetition, of a looping do-group and to proceed with the next iteration of a specified do-group.

Syntax



Rules

ITERATE is only valid within a looping do-group.

If the labeld operand is specified, it must be a label of the DO statement of a containing looping do-group. Control proceeds to the DO statement of the do-group that the labeld operand specifies. This do-group must be in the same procedure (internal or main) as the ITERATE statement.

If the labeld operand is omitted, control proceeds with the next iteration of the do-group that immediately contains the ITERATE statement.

If the exit condition of the do-group to which the ITERATE statement returns is met, control leaves that do-group and passes to the next sequential instruction.

The word ITERATE cannot be used as a compile statement if it is used elsewhere as a variable name.

Example

In the following example, if a specified condition is true, the current iteration of the do-group ends, and execution continues at LABEL1, the DO statement named after the ITERATE statement. If LABEL1 were not specified after the ITERATE statement, control would pass to LABEL2, the DO statement that immediately contains the ITERATE statement.

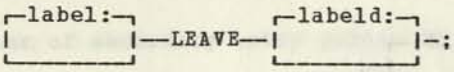
```
LABEL1:  DO I = 1 TO 10;
           .
           .
           .
   LABEL2:  DO J = 1 TO 10;
              .
              .
              .
              .
     IF X < 0
     THEN ITERATE LABEL1;
              .
              .
   END LABEL2;
           .
           .
END LABEL1;
```

LEAVE -- Exit from a Do-Group

Purpose

The LEAVE statement causes control to leave a do-group as if the do-group had terminated normally.

Syntax



Rules

LEAVE is only valid within a do-group.

If the labeld operand is specified, it must be a label of the DO statement of a containing do-group. The do-group that loses control is the group headed by the DO statement that the labeld operand specifies. This do-group must be in the same procedure (internal or main) as the LEAVE statement.

If the labeld operand is omitted, the do-group that loses control is the group that immediately contains the LEAVE statement.

The word LEAVE cannot be used as a compile statement if it is used elsewhere as a variable name.

Example

In the following example, if a specified condition is true, control leaves the LABEL1 do-group and continues at LABEL4. If LABEL1 were not specified after the LEAVE statement, control would pass to LABEL3, the next executable statement after the end of the LABEL2 do-group that contains the LEAVE statement.

```
LABEL1:  DO I = 1 TO 10;
           .
           .
           .
    LABEL2:  DO J = 1 TO 10;
               .
               .
               .
        IF X > 0
        THEN LEAVE LABEL1;
               .
               .
    END LABEL2;
    LABEL3:  DO K UNTIL A = B;
               .
               .
               .
    END LABEL3;
END LABEL1;
LABEL4:  DO L = 1 TO 10;
```

SAVEREG: Specifies the register to be used to locate save areas to hold copies of registers at inter-module interfaces.

NOSAVEREG: Indicates that no register is to be used to locate save areas.

RETREG: Specifies the register to be used for the BALR instruction in CALL statements and for the matching BR instruction in RETURN statements.

NORETREG: Indicates that there is no return register.

RTOREG: Specifies the register to be used to hold the address of a branch target in a RETURN-TO statement.

NORTOREG: Indicates that no register is to be used to hold the address of a branch target in a RETURN-TO statement.

PARMREG: Specifies the register to be used to locate parameter lists at CALL, PROCEDURE, or ENTRY statements.

NOPARMREG: Indicates that no register is to be used to locate parameter lists at CALL, PROCEDURE, or ENTRY statements.

RCODREG: Specifies the register to be used to pass a return code value back to a CALLing procedure.

NORCODREG: Indicates that no register is to be used to pass a return-code value back to a CALLing procedure.

BRANREG: Specifies the register to be used to hold the address of CALLed entry points.

NOBRANREG: Indicates that no register is to be used to hold the address of CALLed entry points.

WORKREGS: Specifies the registers to be avoided by the global register assignment phase.

NOWORKREGS: Indicates that the global register assignment phase can use all non-restricted registers.

PATCHAREA: Allocates a storage area for use as a maintenance space within the generated code.

NOPATCHAREA: Indicates that no storage area is allocated for use as a maintenance space within the generated code.

CODEBNDRY: Indicates that the code being compiled starts at the specified boundary at execution time.

NOCODEBNDRY: Indicates that the code being compiled starts at location 0 at execution time.

MAIN -- Procedure Statement Option

The MAIN option is coded as:

```
--------------------MAIN---------------------
```

MAIN specifies that no registers are saved on initial entry to the procedure, and that a RETURN or the corresponding procedure END statement corresponds to either: (a) a loop stop of the form "J *", for LINKAGE(1) or LINKAGE(2); or (b) a DPPX "EXIT" function for LINKAGE(3).

Parameters may be passed to a MAIN procedure. It is the user's responsibility to ensure that PARMREG points to an appropriate parameter address list.

If MAIN is specified, the options SAVE and NOSAVE are invalid.

REENTRANT -- Procedure Statement Option

The REENTRANT option is coded as:

```
------------------REENTRANT----------------------
```

When REENTRANT is specified as an option on a PROCEDURE statement, the compiler generates reentrant code. The external procedure and its internal procedures are reentrant as a unit; internal procedures are not separately reentrant. Storage for AUTOMATIC data in the external and internal procedures is obtained as one block at the beginning of the procedure, and freed at the end.

REENTRANT has an effect on procedure option defaults and declaration defaults. If REENTRANT is specified, the default storage class for declarations is AUTOMATIC, and the procedure options AUTOREG(28) and NOSTATREG are assumed by default.

If REENTRANT is not specified, and if it is not implied by specifying one or more of the options AUTOREG, AUTODATA, NOAUTOREG, or NOAUTODATA, then the default storage class for data is STATIC LOCAL.

LINKAGE -- Procedure Statement Option

The LINKAGE option is coded as:

```
------------LINKAGE-----(---number---)-------
```

This option indicates which linkage conventions should be observed by the compiler.

LINKAGE(0): Indicates that the compiler should not generate any prologue or epilogue code at all. No CSECT card will be generated, but the end of the generated assembler code for the module will be marked by an assembler END statement. It is the user's responsibility to ensure that any dynamic storage required is obtained and that any required or defaulted AUTOREGs or STATREGs are loaded with correct values. This option should not be specified for code that will be used with the DPPX operating system.

LINKAGE(1): Indicates that the compiler should generate a CSECT card for the main name of the procedure, and ENTRY cards for any secondary names, but should not generate any code to save or restore registers, or to provide addressability to STATIC or AUTOMATIC data. It is again the user's responsibility to ensure that any dynamic storage required is obtained, and that registers are loaded with correct values.

See Figure 14 for the register conventions that are in effect for LINKAGE(1).

LINKAGE(2): Is the default value. The compiler saves and restores registers, and provides addressability to data areas.

See Figure 14 for the register conventions that are in effect for LINKAGE(2).

LINKAGE(3): Simplifies intra-modular linkage in code for execution under DPPX Base. It allows several additional procedure options, such as: GETAUTO, USESTACK, SUBPOOL, and EID.

The LINKAGE(3) option imposes its own set of register conventions. See Appendix E for details.

PARMREG -- Procedure Statement Option

The PARMREG option is coded as:

```
--------------PARMREG(register)--------------------
```

This option specifies the register to be used to locate parameter address lists at CALL, PROCEDURE, and ENTRY statements. The register must be an even-numbered register and must be 0.

Note: LINKAGE(3) generates prolog and epilog code to invoke linkage assist routines that are dependent on this register number. Therefore, changing the register number means that the user must replace the linkage assist routines.

NOPARMREG -- Procedure Statement Option

The NOPARMREG option is coded as:

```
----------------NOPARMREG----------------------------
```

This option specifies that no register is to be used to locate parameter address lists. This is invalid if any CALL, PROCEDURE, or ENTRY has a parameter list.

BRANREG -- Procedure Statement Option

The BRANREG option is coded as:

```
-------------BRANREG(register)-------------------------
```

This option specifies the register to be used as a branch register. This register is used to hold the address of the target entry of a CALL statement. The register must be a primary even-numbered register and must not be 0.

If LINKAGE(3) is coded the default is 8. For other values of LINKAGE, the register is defaulted to 10.

Note: LINKAGE(3) generates prolog and epilog code to invoke linkage assist routines that are dependent on this register number. Therefore, changing the register number means that the user must replace the linkage assist routines.

NOBRANREG -- Procedure Statement Option

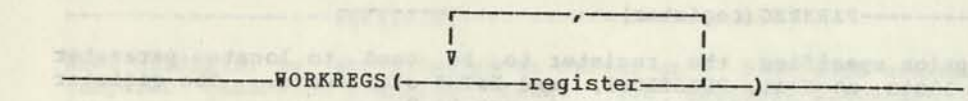
The NOBRANREG option is coded as:

```
----------------NOBRANREG---------------------------------
```

This option indicates that no register is to be used as a branch register.

WORKREGS -- Procedure Statement Option

The WORKREGS option is coded as:

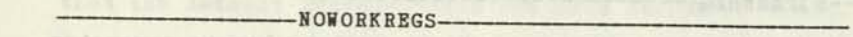


This option specifies one or more registers that are not to be used by the register assignment algorithm during optimization. These registers are reserved for the compile phase's final code generation phase.

This option is not normally needed. It is supplied for the use of programmers who have a stringent optimization problem.

NOWORKREGS -- Procedure Statement Option

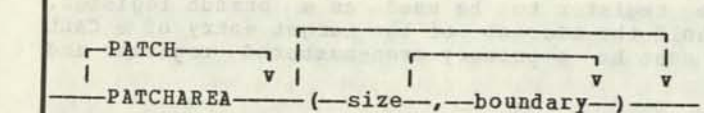
The NOWORKREGS option is coded as:



This option indicates that the register assignment phase of optimization is to use all non-restricted registers. This option should be used with care. It can lead to more highly optimized code, but it exposes the user to the possibility of a failure-to-compile situation because code generation cannot find any registers to evaluate expressions in.

PATCHAREA -- Procedure Statement Option

The PATCHAREA option is coded as:



The PATCHAREA option causes the compiler to allocate a storage area for use as a maintenance space within the generated code. The argument size is a percentage of the storage size needed for the generated code. For example, if size were specified as 10 when 1500 bytes of code are generated, the compiler adds 150 bytes as a patch area. The value of size must be expressed as an integer in the range 1-99. Its default value is 5.

If the argument boundary is coded, the patch areas are embedded in the middle of the executable instructions. A patch area immediately precedes every occurrence of boundary. The value of boundary must be expressed as a decimal integer. If boundary is omitted, a single patch area is allocated before the STATIC data and immediately after the executable instructions.

When, as in most cases, code remains beyond the last patch area caused by the boundary argument, another patch area consisting of size percent of this residue is inserted immediately after the code. This additional patch area may be eliminated by coding the NORESIDUE procedure option.

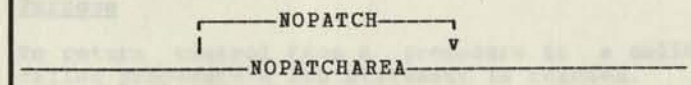
An example of the results of coding the PATCHAREA procedure option follows. If PATCHAREA(10,2000) is coded and 2400 bytes of instructions are generated, the compiler allocates maintenance space as follows:

```
1800 bytes of code
 200 bytes of patch area
 600 bytes of code
  60 bytes of patch area
```

Each patch area begins with a 10-character constant, PATCH AREA. Because of the resolution of long/short jumps, any patch area embedded in the code may be slightly smaller than the designated size.

NOPATCHAREA -- Procedure Statement Option

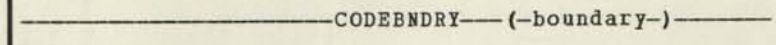
The NOPATCHAREA option is coded as:



The NOPATCHAREA option causes the compiler not to allocate a storage area for use as a maintenance space within the generated code. If neither PATCHAREA nor NOPATCHAREA is coded, NOPATCHAREA is the default.

CODEBNDRY -- Procedure Statement Option

The CODEBNDRY option is coded as:

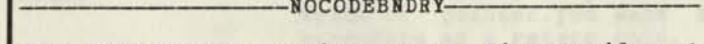


The CODEBNDRY option causes the compiler to assume that the code being compiled starts at the specified boundary at execution time. This option can interact with the PATCHAREA option, which normally assumes that code begins at location 0. The value of the boundary argument must be a decimal integer.

Note: The CODEBNDRY option causes the compiler to generate code based on an assumption. The user is responsible to ensure that the code has the proper origin. He can do this by using the appropriate linkage-editor controls.

NOCODENBNDRY -- Procedure Statement Option

The NOCODENBNDRY option is coded as:



The NOCODENBNDRY option causes the compiler to assume that the code being compiled starts at location 0 at execution time. If neither the CODEBNDRY nor the NOCODEBNDRY option is coded, NOCODEBNDRY is the default.

RETURN -- Return Control Back to Calling Procedure

Purpose

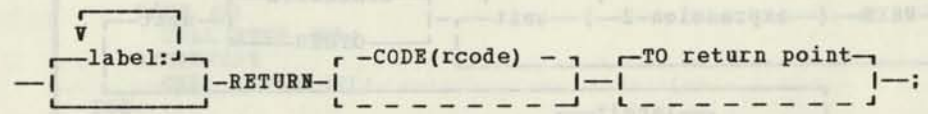
To return control from a procedure to a calling procedure before the called procedure's END statement is reached.

Rules

You can cause a return either to the statement following the CALL statement or to some other labeled statement.

A value can be returned to the calling procedure. RETURN is a reserved keyword; do not use it as a variable name.

Syntax



Operands

label — Optional. Code one or more labels, each followed by a colon.

CODE — A value is to be returned to the calling procedure.

rcode — The rcode value, in parentheses, is a an arithmetic value or pointer you want returned to the calling procedure as a return code.

This value is returned to the calling procedure in RCODEREG (PROCEDURE option).

Code a variable, a constant, or a complex expression. Some examples of the value are:

```
CODE(12)          CODE('80'X)
CODE(P -> RET)    CODE((S+4)*X)
CODE(ADDR(A))
CODE(R3 + R4)
```

TO — Code is returned to a specified return point, not to the statement after CALL.

return point — The return point is a name that identifies the statement you want to return to. It must be a label.

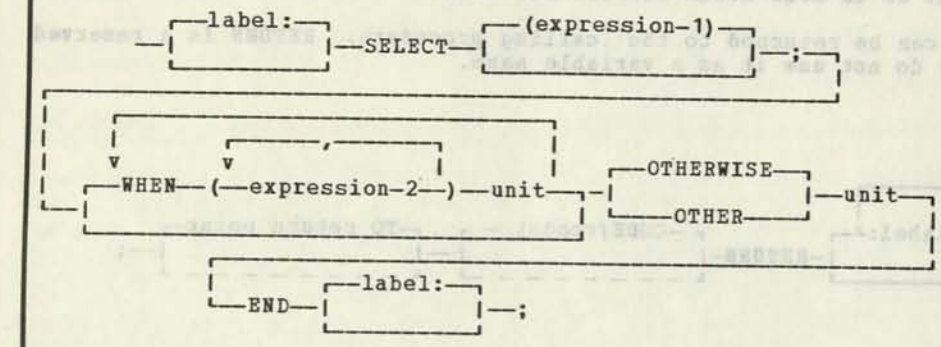
If you do not include the keyword TO and a return point variable on the RETURN statement, control is returned to the statement following the CALL statement in the calling procedure.

SELECT -- Chooses Execution Path

Purpose

The SELECT statement allows the selection of one execution path from a set of mutually exclusive choices. The SELECT statement begins the select-group; the END statement completes it.

Syntax



Rules

A select-group can appear in any context in which a do-group is permitted. A select-group must be terminated by a matching END statement.

Each unit is a single statement, do-group, or select-group.

If the expression-1 argument is omitted, then each expression-2 must be a valid relational expression. If expression-1 is supplied, expression-2 is treated as if the relational expression, expression-1 = expression-2, were coded.

Each relational expression-2 is evaluated in the order written until one is found to be true. When and if this occurs, control passes to the unit associated with the WHEN clause containing the expression-2. If no such expression-2 is found to be true, control passes to the unit associated with the OTHERWISE.

After execution of a unit following a WHEN or OTHERWISE clause, control passes to the first executable statement following the select-group, unless normal flow of control is altered within the unit, for example, by a GOTO.

WHEN or OTHERWISE keywords can not have a label prefix. However, the associated units may have label prefixes. The keywords SELECT, WHEN, OTHERWISE, and OTHER can not be used as compile statements if they are used elsewhere as variable names.

Example:

In the following example, the select-group chooses an execution path based upon the type of record that it is passed:

```
DCL P PTR;
DCL 1 RECORD BASED (P),
    2 RECTYPE FIXED (8),
      .
      .
      .   ;
CALL READ (P);              /* Sets "P" with addr of record */
DO WHILE (P¬=0);
   SELECT (RECTYPE);
     WHEN (1)
       CALL RTN1 (P);
     WHEN (2,5)             /* Ignore these types */
       ;
     WHEN (3)
       DO;
         .
         .
         .
       END;
     WHEN (4)
       CALL RTN4 (P);
     OTHERWISE
       CALL ERROR (P);
  END;
  CALL READ (P);            /* Get next record */
END;
```

BUILT-IN FUNCTIONS FOR COMPILE STATEMENTS

The built-in functions are aids to coding compile statements. Some of them allow you to do things that cannot be done using compile instructions. Others are a compact way of performing operations that would require several statements. In most cases a built-in function is like a variable, returning a value to the place in a statement where it appears, for use by the remainder of the statement.

The keywords that identify the built-in functions are not reserved. However, you cannot use any of these keywords as both a built-in function and a variable name. If the keyword is not explicitly declared elsewhere as user data, the compiler interprets it as a built-in function.

The built-in functions are summarized in Figure 16.

Built-In Function	Use
ABS	Return absolute value (positive, unsigned) of specified variable or expression.
ADDR	Return address of specified non-register variable, or reserve storage in the STATIC data area for a specified constant.
DIM	Return extent (size) of a specified dimension in a specified array.
EVAL	Return the value of the result of the specified expression.
LENGTH	Return length of specified constant or variable.
MAX	Return the maximum value from the specified expressions.
MIN	Opposite of MAX.

Figure 16. Built-In Functions and Their Uses

not declared as a register, it may be assigned to one, and the compiler-generated code for the machine instruction will be adjusted accordingly.

Instructions That Require Registers

The following mnemonics are exceptions to the above discussion. They each require a variable declared as a register in the operand listed:

BAL	operand-1
BALR	operand-1
BCTR	operand-1
DHR	operand-1
JAL	operand-1
LHQ	operand-2
MHR	operand-1
STHQ	operand-2

Notes:

1. The MHR instruction has different meanings depending on whether or not the first register number is a multiple of four. See the description in the publication, IBM 8100 Information System: Principles of Operation.

2. The DHR instruction's first register number must be a multiple of four, and both this register and the even-numbered register following it must be restricted and then loaded using conventional assignment statements before issuing the DHR.

Extended Mnemonics

The Control-Immediate and Input/Output instructions have extended mnemonics (for example: KIR and KIW) that indicate whether the instruction is performing a read or a write. A "read" causes a value to materialize in a register, which implies that store code follows the supported machine instruction. A "write" implies that a value exists in a register to be written and therefore there must be code to load a register prior to issuing the supported machine instruction.

Note: The compiler's optimization phase frequently assigns the given expression to a register so that load/store code is not needed.

COMPILER ADJUSTMENTS TO VARIABLES IN ARGUMENTS

If the supported machine instruction requires an address (such as BCR), the compiler generates instructions to load the address of the variable you specify into a register.

Apart from loading and storing, the compiler makes no adjustments to the variables specified. In particular, the compiler performs no padding or truncation of variables.

If the instruction requires an indirect operand (such as LHNI), code a pointer to the item. The compiler causes this pointer to be adjusted if the instruction increments or decrements.

SUPPORTED MACHINE INSTRUCTIONS AND EXTENDED MNEMONICS

The following table (Figures 18 and 19) lists the supported machine instructions, operands, and names.

Note that the compiler supports some extended mnemonics (for example, BER and KIR) in addition to the actual machine instructions.

The following explains the operand symbols used in the table below.

Each operand will be a valid PL/DS expression, but will have constraints on its attributes.

Operand	Meaning
B	operand must be FIXED(8) or CHAR(1) or BIT(8) on a byte boundary
E	operand must be ENTRY
H	operand must be FIXED(15) or FIXED(16) or CHAR(2) or BIT(16) on a halfword boundary
In	operand must be immediate value (which can be represented in n bits)
L	operand must be LABEL
P	operand must be POINTER or FIXED(32)
RS	operand must be FIXED(16) or BIT(16) (user has responsibility to ensure that the operand contains a valid register space address)
Z	operand must be the constant 0
*	operand attributes are not checked
xA	operand must be accessible via address-in-register
xD	operand must be accessible via base-displacement
xL	operand must have attribute LOCAL
xO	operand is an output which implies that neither constants nor arithmetic expressions nor string expressions are permitted.
xR	operand must be declared as register
xU	operand must be unsigned
xZ	zero used if explicitly specified (not a value of zero to be associated with some register)
cc	opcode is one of extended branch mnemonics:

Z	E	O	TE
P	H	X	null
M	L	NO	NM
NZ	NE	NX	
NP	NH	NL	
Y	V		

Note: The control immediate (KI) machine instruction must be used with care. See the publication, 8100 Information System: Principles of Operation.

The PL/DS compiler does not recognize or interpret the specialized meanings of control immediate operands, and does not act on the values preceding and following the control immediate statement in a "knowing" way.

INDEX

SC27-0446-0

Dist Proc Devel Sys Prog Language for Dist Systems (File No. 8100/S370/4300-31) Printed in U.S.A. SC27-0446-0

International Business Machines Corporation
Data Processing Division
1133 Westchester Avenue, White Plains, N.Y. 10604

IBM World Trade Americas/Far East Corporation
Town of Mount Pleasant, Route 9, North Tarrytown, N.Y., U.S.A. 10591

IBM World Trade Europe/Middle East/Africa Corporation
360 Hamilton Avenue, White Plains, N.Y., U.S.A. 10601