TPF Database Facility Release 1
Transaction Processing Facility Version 4.1

Structured Programming Macros

SH31-0183-04

TPF Database Facility Release 1
Transaction Processing Facility Version 4.1

Structured Programming Macros

SH31-0183-04

Note!
Before using this information and the product it supports, be sure to read the general information under “Notices”.

Fifth Edition (October 2001)

This is a major revision of, and obsoletes, SH31-0183-03.

This edition applies to Version 1 Release 1 Modification Level 3 of IBM Transaction Processing Facility Database Facility, program number 5706-196, and Version 4 Release 1 Modification Level 0 of IBM Transaction Processing Facility, program number 5748-T14, and to all subsequent releases and modifications until otherwise indicated in new editions or technical newsletters. Make sure you are using the correct edition for the level of the product.

Order books through your IBM representative or the IBM branch office serving your locality. Books are not stocked at the address given below.

IBM welcomes your comments. Address your comments to:

IBM Corporation
TPF Systems Information Development
Mail Station P923
2455 South Road
Poughkeepsie, NY 12601-5400
USA

Contents

Figures

Tables

About This Book

This book describes two sets of structured programming macros (SPMs). One set of SPMs is provided with the TPF Database Facility (TPFDF) product and the Transaction Processing Facility (TPF) system and is referred to as the TPFDF SPMs. The other set of SPMs is provided with the TPF system only and is referred to as the TPF SPMs.

In this information, abbreviations are often used instead of spelled-out terms. Every term is spelled out at first mention followed by the all-caps abbreviation enclosed in parentheses; for example, structured programming macro (SPM). Abbreviations are defined again at various intervals throughout the book. In addition, the majority of abbreviations and their definitions are listed in the master glossary in *TPFDF Glossary* and *TPF Glossary*.

Before You Begin

Before using this book:

1. Read "Structured Programming Macros Introduction" on page 3 and determine the set of SPMs that you want to use.
2. If you are going to use the TPFDF SPMs, see Part 2, "TPFDF Structured Programming Macros" on page 7.
3. If you are going to use the TPF SPMs, see Part 3, "TPF Structured Programming Macros" on page 87.

Who Should Read This Book

This book is intended for application programmers who are currently working with one of the following:

- Transaction Processing Facility (TPF) system and IBM High Level Assembler/MVS & VM & VSE (HLASM)
- Airline Control System (ALCS), also referred to as TPF/MVS, and IBM Assembler H or IBM High Level Assembler/MVS & VM & VSE (HLASM).

How This Book Is Organized

This book has 3 parts as follows:

Part 1, "Structured Programming Macros Overview"
Provides an overview of structured programming and contains some general rules for using the structured programming macros (SPMs).

Part 2, "TPFDF Structured Programming Macros"
Provides detailed information about the TPFDF SPMs.

Part 3, "TPF Structured Programming Macros"
Provides detailed information about the TPF SPMs.

For your convenience, a set of tabs (GH31-0184) is available for this book. These tabs help you to quickly access the major sections of this book.

Conventions Used in This Book

This book uses the following conventions:

Typography	Examples of Usage
italic	Used for important words and phrases. For example: A *database* is a collection of data. Used to represent variable information. For example: Enter **ZUDFC DISPLAY ID-***fileid*, where *fileid* is the file identifier (ID) of the file for which you want statistics.
bold	Used to represent keywords. For example: Enter **ZUDFC HELP** to obtain help information for the ZUDFC command.

`monospaced`	Used for messages and information that displays on a screen. For example:
	`PROCESSING COMPLETED`
	Used for C language functions. For example:
	`dfcls`
	Used for examples. For example:
	`ZUDFC DISPLAY ID-J5`

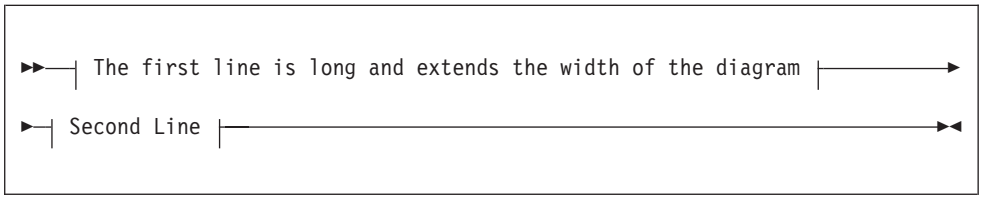
bold italic	Used for emphasis. For example: You ***must*** type this command exactly as shown.
CAPital LETters	Used to indicate valid abbreviations for keywords. For example: KEYWord=*option*

How to Read the Syntax Diagrams

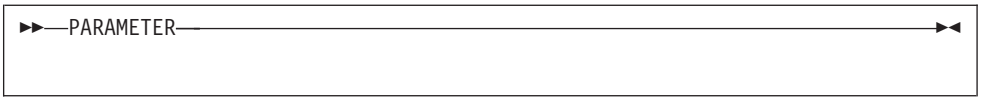
This section describes how to read the syntax diagrams (informally called *railroad tracks*) used in this book.

- Read the diagrams from left-to-right, top-to-bottom, following the main path line. Each diagram begins on the left with double arrowheads and ends on the right with 2 arrowheads facing each other.

- If a diagram is longer than one line, the first line ends with a single arrowhead and the second line begins with a single arrowhead.



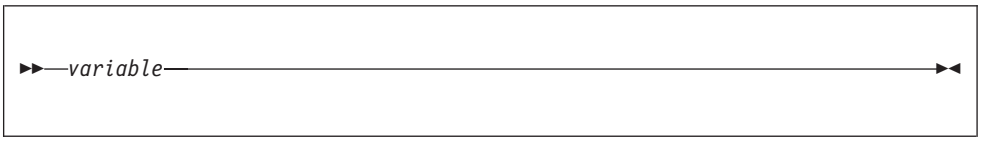
- A word in all uppercase is a parameter that you must spell ***exactly*** as shown.



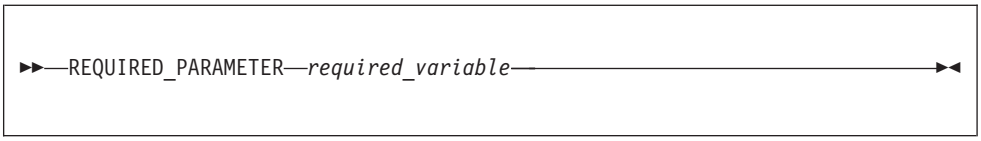
- If you can abbreviate a parameter, the optional part of the parameter is shown in lowercase. (You must type the text that is shown in uppercase. You can type none, one, or more of the letters that are shown in lowercase.)

  Note: Some TPF commands are case-sensitive and contain parameters that must be entered exactly as shown. This information is noted in the description of the appropriate commands.

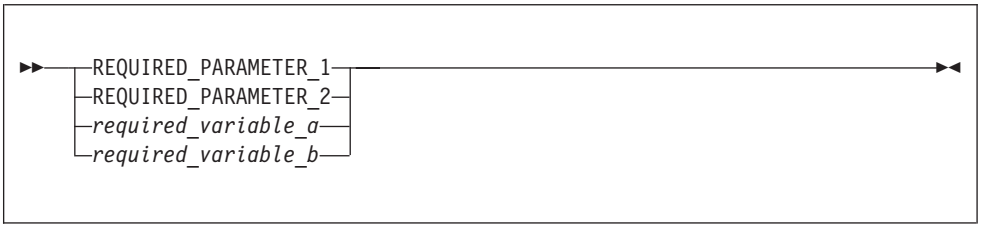
- A word in all lowercase italics is a *variable*. Where you see a variable in the syntax, you must replace it with one of its allowable names or values, as defined in the text.



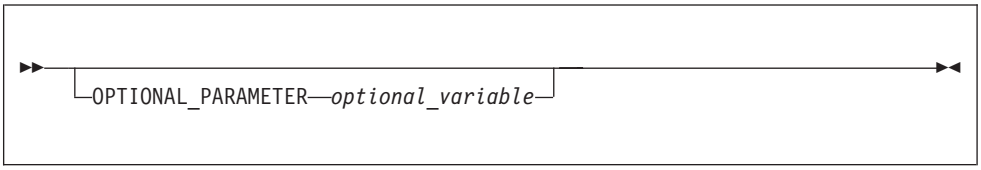
- Required parameters and variables are shown on the main path line. You must code required parameters and variables.



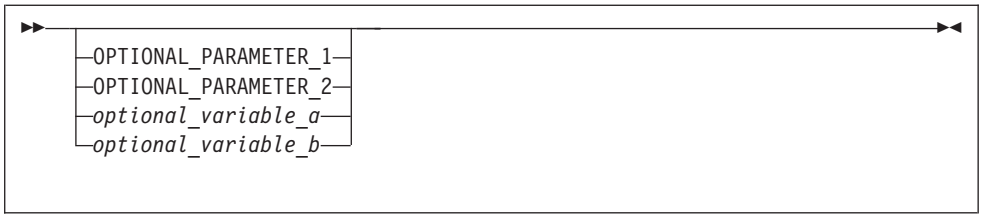
- If there is more than one mutually exclusive required parameter or variable to choose from, they are stacked vertically.



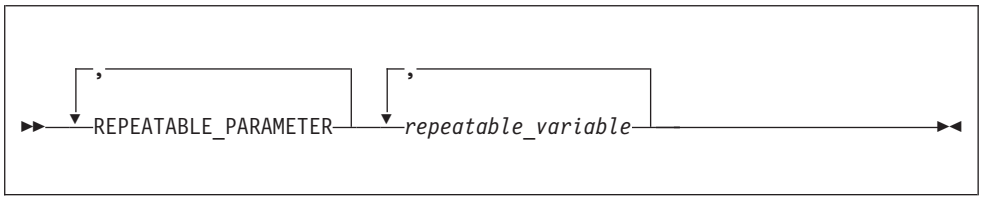
- Optional parameters and variables are shown below the main path line. You can choose not to code optional parameters and variables.



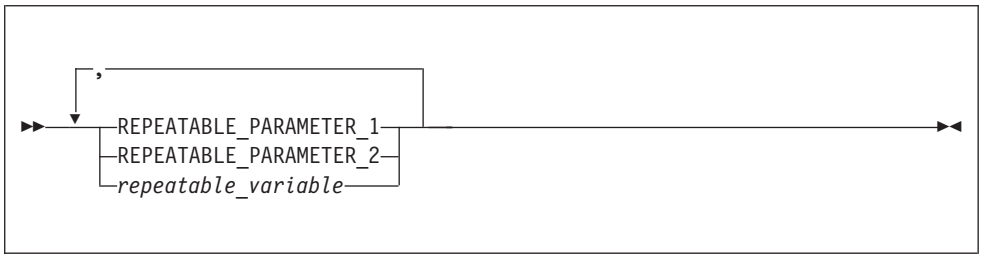
- If there is more than one mutually exclusive optional parameter or variable to choose from, they are stacked vertically below the main path line.



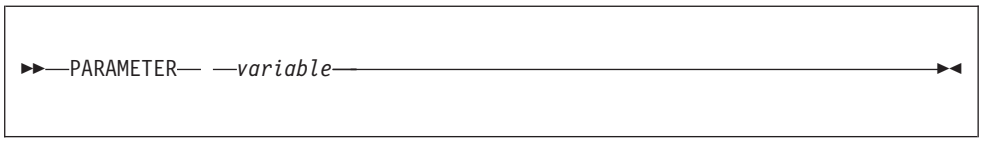
- An arrow returning to the left above a parameter or variable on the main path line means that the parameter or variable can be repeated. The comma (,) means that each parameter or variable must be separated from the next parameter or variable by a comma.



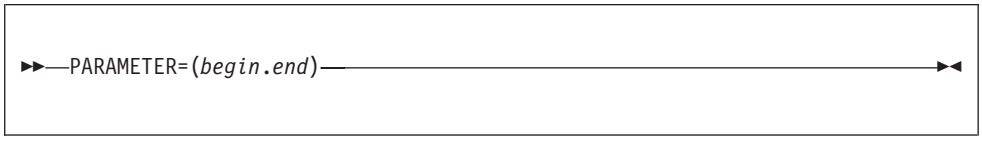
- An arrow returning to the left above a group of parameters or variables means that more than one can be selected, or a single one can be repeated.



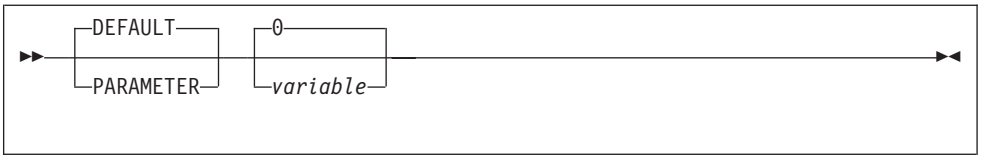
- If a diagram shows a blank space, you must code the blank space as part of the syntax. In the following example, you must code **PARAMETER** *variable*.



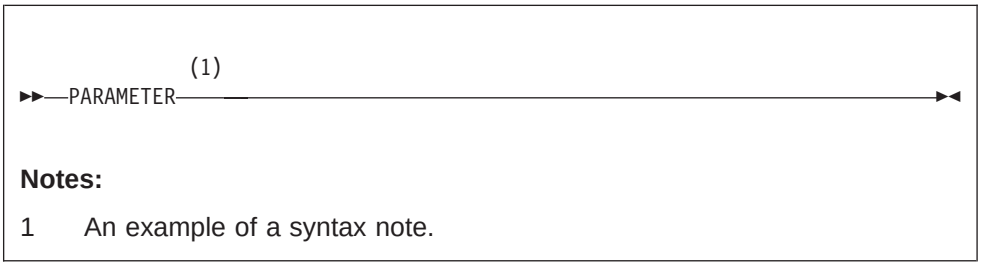
- If a diagram shows a character that is not alphanumeric (such as commas, parentheses, periods, and equal signs), you must code the character as part of the syntax. In the following example, you must code **PARAMETER=(***begin***.***end***)**.



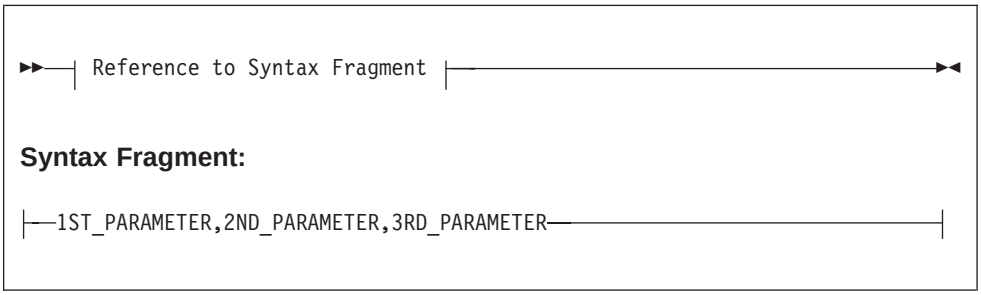
- Default parameters and values are shown above the main path line. The TPF system uses the default if you omit the parameter or value entirely.



- References to syntax notes are shown as numbers enclosed in parentheses above the line. Do not code the parentheses or the number.



- Some diagrams contain *syntax fragments*, which serve to break up diagrams that are too long, too complex, or too repetitious. Syntax fragment names are in mixed case and are shown in the diagram and in the heading of the fragment. The fragment is placed below the main diagram.



Related Information

A list of related information follows. For information on how to order or access any of this information, call your IBM representative.

IBM TPF Database Facility (TPFDF) Books

- *TPFDF Database Administration*, SH31-0175
- *TPFDF Programming Concepts and Reference*, SH31-0179.

IBM Transaction Processing Facility (TPF) 4.1 Books

- *TPF General Macros*, SH31-0152.

IBM Airline Control System (ALCS) Books

- *ALCS Application Programming Reference — Assembler Language*, SH19-6949.

Miscellaneous IBM Books

- *ESA/370 Principles of Operation*, SA22-7200
- *ESA/390 Principles of Operation*, SA22-7201
- *High Level Assembler for MVS & VM & VSE Language Reference*, SC26-4940.

Online Information

- *TPFDF Commands*
- *TPFDF Glossary*
- *TPFDF Messages (System Error, Online, Offline)*
- *TPFDF Utilities*.

How to Send Your Comments

Your feedback is important in helping to provide the most accurate and highest quality information. If you have any comments about this book or any other TPF information, use one of the methods that follow. Make sure you include the title and number of the book, the version of your product and, if applicable, the specific location of the text you are commenting on (for example, a page number or table number).

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

- If you prefer to send your comments electronically, do either of the following:
  - Go to http://www.ibm.com/tpf/pubs/tpfpubs.htm.

    There you will find a link to a feedback page where you can enter and submit comments.
  - Send your comments by e-mail to tpfqa@us.ibm.com
- If you prefer to send your comments by mail, address your comments to:

  IBM Corporation
  TPF Systems Information Development
  Mail Station P923
  2455 South Road
  Poughkeepsie, NY 12601-5400
  USA
- If you prefer to send your comments by FAX, use this number:
  - United States and Canada: 1 + 845 + 432 + 9788
  - Other countries: (international code) + 845 + 432 +9788

Part 1. Structured Programming Macros Overview

Structured Programming Macros Introduction

The TPF Database Facility structured programming macros (SPMs) and Transaction Processing Facility (TPF) SPMs are used to add *structured programming* verbs to existing assembler language. Both sets of SPMs provide the basic structured programming constructions, such as:

- Selection, for example:

  ```
  IF condition then process A ELSE process B
  ```

- Iteration, for example:

  ```
  DO WHILE condition
  DO UNTIL condition
  ```

In addition, the TPFDF SPMs include some additional functions, such as macros that allow you to convert between data types. See “TPFDF Structured Programming Macros General Information” on page 9 for more information about the additional functions that are included with the TPFDF SPMs.

- If you are writing application programs that will run in a TPF environment, you can use the TPFDF SPMs or the TPF SPMs.
- If you are writing application programs that will run in an Airline Control System (ALCS) environment, you can only use the TPFDF SPMs.

Recommendations for TPF Application Programmers

- The TPFDF SPMs offer more functions and, in many cases, are easier to use. If you do not already have application programs that are written using the TPF SPMs, consider using only the TPFDF SPMs.
- If you do use both sets, avoid mixing the TPFDF SPMs and the TPF SPMs in one application program.

Advantages of Structured Programming Macros

Structured application programs require more discipline at the design and logical structuring stage but they can be coded more quickly. The time required to reach the testing stage is roughly the same, but the benefits of SPMs are significant from this point and onward.

Using the SPMs requires a more disciplined approach to programming and, therefore, provides the following advantages:

- Application programs are easier to read and understand.
- Application programs are less likely to contain logic errors.
- Errors are more easily found.
- Higher productivity during application program development.
- Improved application program design.
- Application programs are more easily maintained.

Using structured programming emphasizes:

- The total logic of an application program.
- What the application program does.
- What the logical flow through the application program should be.

You must consider these aspects before writing detailed instructions. This leads to a clearer application program structure, improved understanding, and a better chance of identifying possible abnormal or error conditions that should be corrected.

Using structured programming techniques also makes it easier for a new application programmer to:
- Read and understand an existing application program.
- Identify modifications for additional functions or correcting errors.

An Overview of Structured Forms

In structured programming theory, only three basic control logic structures are required to program any function:
- Sequences
- Selection
- Iteration.

A *sequence* is the processing of one function after the other, as shown in Figure 1. No special logic is required because processing is always sequential.

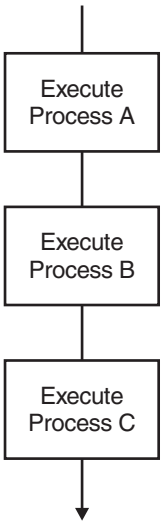


Figure 1. Sequence: Processing Code Sequentially

A *selection* is the choice between two or more functions to be processed based on a condition.

For example, the IF macro group, shown in Figure 2 on page 5, is used for selection.

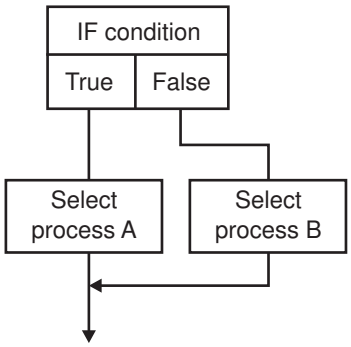


Figure 2. Selection: Using a Condition

The CASE macro group, shown in Figure 3, is another example of a macro group used for selection.

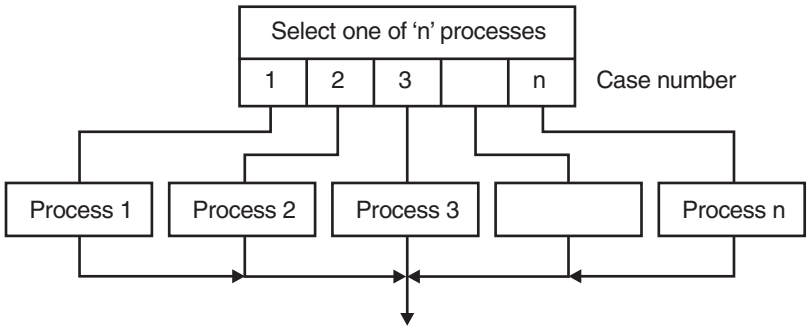


Figure 3. Selection: Using a Case Number

An *iteration* is the repeated processing of the same code while, or until, a condition is true, as shown in Figure 4.

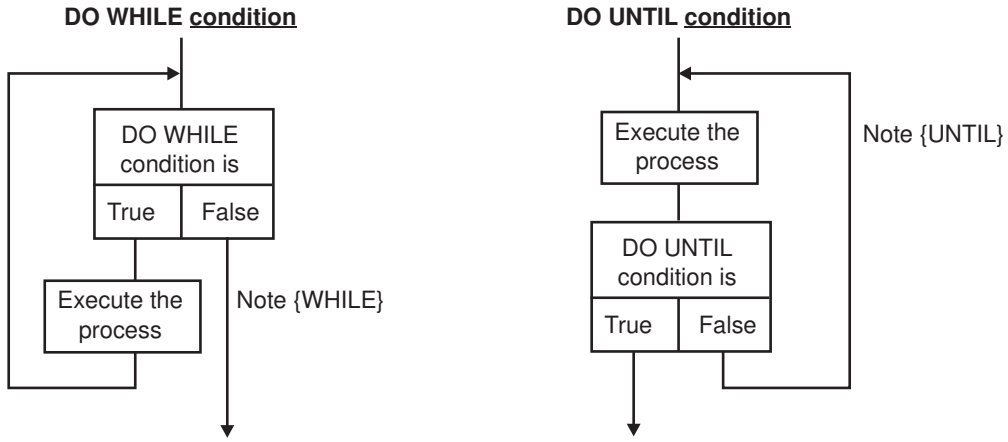


Figure 4. Simple Iteration: The Difference between DO WHILE and DO UNTIL

Notes:

1. The code can be bypassed (never processed) if *condition* is false on entry to the DO WHILE loop.
2. The code is processed at least once before the DO UNTIL *condition* is tested.

General Rules for Structured Programming Macros

Use the following guidelines when using SPMs.

- Where possible, code only one entry and one exit point.

  The following methods of changing program flow are exceptions to the one entry point and one exit point rule:
  - Multiple ENTNC macro calls
  - Multiple ENTDC macro calls
  - Error exit routines
  - Transfer vector entry points.
- If you use SPMs in your application program, use the SPMs exclusively; that is, do not mix the SPMs with basic assembler test-and-branch coding.
- To improve readability and debugging, ensure that complete logic structures (IF—END-IF, DO—END-DO, and so on) are, where possible:
  - Confined to 1 page
  - No longer than 2 pages.

  Use subroutines as appropriate to do this.
- Include comments.

Part 2. TPFDF Structured Programming Macros

TPFDF Structured Programming Macros General Information

The following provides the following general information about the TPFDF structured programming macros (SPMs):

- Additional functions available with TPFDF SPMs
- Nesting levels and indenting in the TPFDF SPMs.

Additional Functions

As mentioned previously, the TPFDF SPMs provide additional functions in addition to the standard structured programming constructions. Some of these additional functions are provided through macros; others are provided through special parameters that you can use in a conditional expression for the basic SPMs (#IF, #DO, and so on).

See “Structured Programming Macros Conditional Expressions” on page 13 for details about conditional expressions, including information about these special parameters. See “TPFDF Structured Programming Macros: Reference” on page 27 for details about all the TPFDF SPMs.

The following briefly describes some of the additional functions available with the TPFDF SPMs.

Function	Description
Converting data	The conversion macros allow you to convert data between data types. See Table 1 on page 10 for a summary of these macros.
Stepping functions	The *step* macros (#STPC, #STPH, and so on) allow you to increment or decrement the contents of a specified location.
Line continuation	The line continuation macro (#) allows you to concatenate complex conditional expressions.
Testing TPFDF return codes	For application programs that are written for TPFDF, there are special parameters that provide a short form method for testing the return conditions in the SW00RTN bits. These parameters allow you to generate consistent code to test return conditions. See *TPFDF Programming Concepts and Reference* for more information about testing return conditions and SW00RTN.
Using TPF and ALCS macros	There are parameters that allow you to use certain TPF and ALCS macros in a conditional expression.
Changing link-label prefixes	Some SPMs generate a *link-label*, which is an internal label to indicate branches in a sequential flow. These macros also have a PREFIX parameter that allows you to change the standard label prefix to improve readability.

Conversion Macros

Table 1 summarizes the conversion macros and provides an example of a string before and after the conversion.

Note: The results shown in this table are also based on other parameters being specified a certain way. See the information for each macro in “TPFDF Structured Programming Macros: Reference” on page 27 for details about all the parameters associated with each macro.

Table 1. Conversion Macro Summary

Macro	Description	Before Conversion	After Conversion
#CONB	Character decimal to binary	C"12713971"	X'C1FFF3'
#COND	Binary to character decimal	X'0000F394'	C"00062356"
#CONH	Character hexadecimal to binary	C"12713971"	X'12713971'
#CONP	Binary to character hexadecimal with EBCDIC interpretation	X'C1FFF3'	C" AFF 3"
#CONS	Binary to character decimal with zero suppression	X'0000F394'	C"62356"
#CONT	Binary to character binary	F"43"	C"00101011"
#CONX	Binary to character hexadecimal	X'C1FFF3'	C"C1FFF3"

Nesting Levels and Indenting

The TPFDF SPMs indicate the nesting level by using assembler messages.

The following is an example of how these messages indicate the nesting level:

```
+*,1   ccccccccccccccccccc code
       .
+*,2   ccccccccccccccccccc code
       .
+*,3   ccccccccccccccccccc code
       .
+*,2   ccccccccccccccccccc code
       .
+*,1   ccccccccccccccccccc code
```

Note: You can suppress the generation of these nesting level assembler messages with the #SPM macro. See “#SPM–Assembly Output Processing” on page 66 for more information.

The nesting level begins at 0 and is incremented by 1 after any of the following macros:

- #IF
- #DO
- #CAST
- #SUBR.

The nesting level is decremented by 1 after any of the following ending macros:

- #EIF
- #EDO
- #ECAS
- #ESUB.

Consider indenting the assembler messages or the code itself to reflect the nesting levels of the program logic.

You can represent the nesting level by indenting the number produced in the assembler message, but not the code itself, as follows:

```
1      ccccccccccccccccccc code
       .
 2     ccccccccccccccccccc code
       .
  3    ccccccccccccccccccc code
       .
 2     ccccccccccccccccccc code
       .
1      ccccccccccccccccccc code
```

Using this method requires you to postprocess the assembler messages that are generated.

Note: The postprocessing is installation-dependent and is not supplied with the TPFDF product or the TPF system.

You can also represent the nesting level by indenting the code itself. If you do this, use the following guidelines:

- Indent the whole program consistently.
- Because explicitly-coded labels are not required in programs using SPMs, start the nesting level 0 code in column 2.
- Do not exceed nesting level 5.
- Indent three columns for each nesting level.
- Begin all SPMs in a particular structure (at a particular nesting level) in the same column.
- Indent the code three columns to the right after any of the following:
  - #IF
  - #DO
  - #CAST
  - #ELIF
  - #DOEX
  - #CASE
  - #ELSE
  - #EXIF
  - #OREL
  - #ELOP

  Code each of the following, and any subsequent code, three columns to the left:
  - #ELIF
  - #DOEX
  - #CASE
  - #ELSE
  - #EXIF
  - #ECAS
  - #EIF
  - #OREL
  - #ELOP
  - #EDO

  For example, in a #DO structure, begin any of the #DO macros in the same column as the #DO macro statement.

```
#DO   WHILE=(...
   LA   R1,...
   LA   R5,...
   #IF   CR,R5,...
      MVC...
   #EIF
#EXIF  ...
 :
#OREL
 :
#EDO
```

Indent any other code enclosed by that #DO structure to the right.

- When using the #EIFM macro to end multiple #IF structures, code the #EIFM *n* macro at (3 × *n*) columns to the left, where *n* is the number of the #IF structures you are ending. See “#IF Macro Group” on page 63 for more information about the #EIFM macro.

Structured Programming Macros Conditional Expressions

One of the key elements of the selection and iteration structured programming structures is the conditional expression.

A *conditional expression* defines the selection criteria for the different functions to be carried out in a macro group. A conditional expression is an expression that evaluates to a true or false value, and the appropriate function is processed based on that value.

Conditional expressions are required parameters for the following macros:
- #IF
- #ELIF
- #DOEX
- #EXIF.

You can also use conditional expressions with the following macros:
- #DO
- #GOTO.

See “TPFDF Structured Programming Macros: Reference” on page 27 for details about these structured programming macros (SPMs).

Forms of Conditional Expressions

The TPFDF SPMs allow a number of different types of conditional expressions:

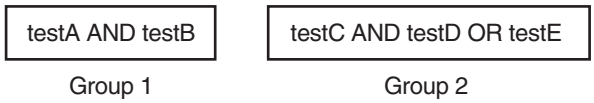
- A symbolic representation of a standard assembler language instruction that results in a condition code setting or branch. This form consists of the following types:
  - Branch on condition code instructions
  - Compare instructions
  - Noncompare instructions.

  See *ESA/370 Principles of Operation* or *ESA/390 Principles of Operation* for more information about specific assembler instructions.
- A symbolic representation of certain TPF or ALCS macro calls.
- Special operands to check a CPU ID.
- Special operands to test the return conditions in the TPFDF SW00RTN bits. This form is available only for application programs written for a TPFDF environment.

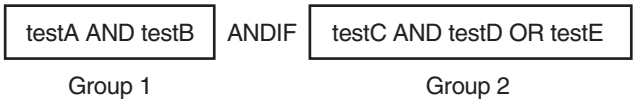
  See *TPFDF Programming Concepts and Reference* for more information about testing return conditions and SW00RTN.

You can also use Boolean connectors to connect individual tests to form complex conditional expressions, as follows:

- Connect individual tests with an AND or OR connector to form groups

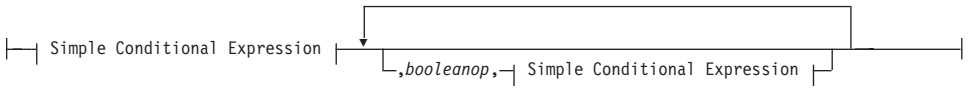


- Connect groups of tests with an ANDIF or ORIF connector to form complex expressions

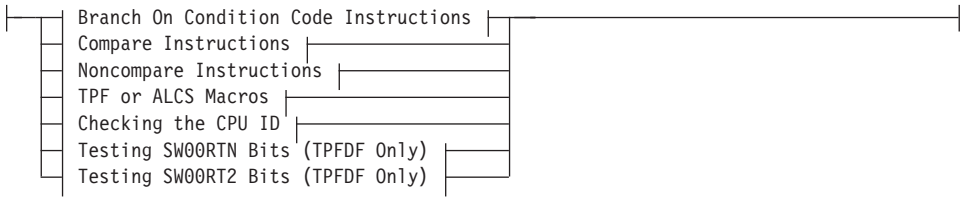


Conditional Expression Format

Conditional Expression:



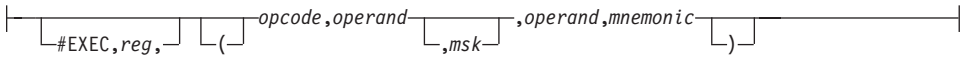
Simple Conditional Expression:



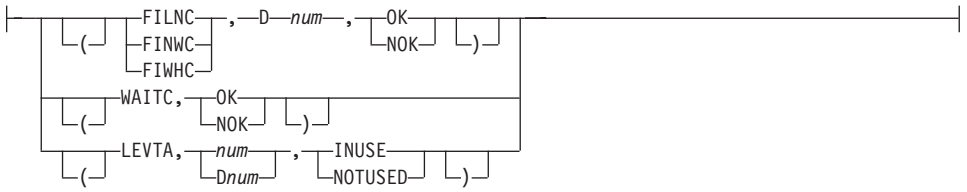
Branch On Condition Code Instructions:

Compare Instructions:

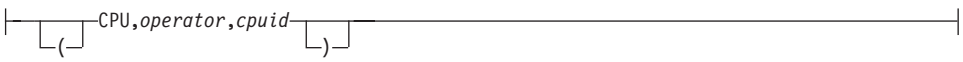
Noncompare Instructions:



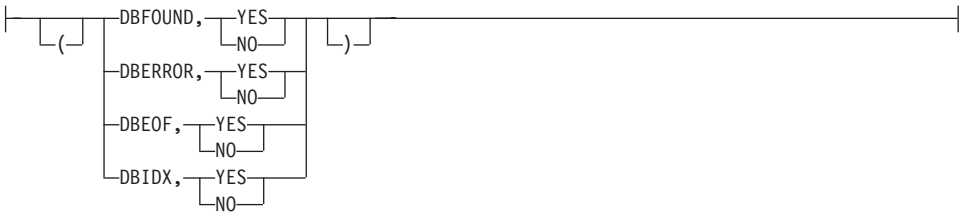
TPF or ALCS Macros:



Checking the CPU ID:



Testing SW00RTN Bits (TPFDF Only):



Testing SW00RT2 Bits (TPFDF Only):

booleanop
is one of the following Boolean connectors:
- AND
- OR
- ANDIF
- ORIF.

n is the condition code mask, in the range 0–15. For example:

```
#IF   (7)
```

Note: If you specify a conditional code mask of 0, a warning MNOTE is issued when the application is assembled.

mnemonic
is one of the condition-code mnemonics shown in Table 2.

Table 2. Mnemonics Allowed in SPM Expressions

	Condition		Complement Condition	
Instruction Type	**Mnemonic**	**Meaning**	**Mnemonic**	**Meaning**
Compare	H	high	NH	not high
	GT	greater than	LE	less or equal
	L	low	NL	not low
	LT	less than	GE	greater or equal
	EQ	equal	NE	not equal
Arithmetic	P POSITIVE	positive	NP NOTPOSITIVE	not positive
	M NEGATIVE	minus (negative)	NM NOTNEGATIVE	not minus (not negative)
	Z	zero	NZ	not zero
	O	overflow	NO	not overflow

Table 2. Mnemonics Allowed in SPM Expressions (continued)

Instruction Type	Condition		Complement Condition	
	Mnemonic	**Meaning**	**Mnemonic**	**Meaning**
Test under Mask	O, ON	ones	NO	not ones
	M MIXED	mixed	NM NOTMIXED	not mixed
	Z ZERO, OFF	zeros	NZ NONZERO	not zeros

#EXEC
generates an EX instruction. See *ESA/370 Principles of Operation* or *ESA/390 Principles of Operation* for more information about the EX instruction.

Note: You can also use the #EXEC macro alone; see "#EXEC–Execute Macro" on page 58 for more information.

reg
is the register to use as the first operand in the EX instruction.

copcode
is an operation code starting with C; for example, CLC, CR, and so on.

opcode
is an operation code that sets a condition code but does not start with C; for example, TM, OC, and so on.

operand
is an operand for the instruction, which can be anything that the assembler language allows for an assembler instruction operand. See *ESA/370 Principles of Operation* or *ESA/390 Principles of Operation* for more information about assembler instructions and valid operands. Some of the types of operands include:

- A register, in the form R*n*, where *n* is the register number; for example, R1.
- A label refers to a defined area of storage (DS), a defined constant (DC), or an equated value. Generally, the size and type of label is implied from the type of instruction. However, you can force the size and type by prefixing the label as follows:

  A*l**label***
  specifies a 4-byte address contained at location *label*.

  F*l**label***
  specifies a fullword starting at location *label*.

  H*l**label***
  specifies a halfword starting at location *label*.

  X*l**label***
  specifies a byte starting at location *label*.

  I*l**label***
  specifies a 1-byte equated value.

  P*l**label***
  specifies packed data at location *label*.
- A numeric value, which must be an integer.
- A literal, for example:
  – A fullword (=F"1000")

  - A halfword (=H"10")
  - Characters (=C"HELLO").

  See *ESA/370 Principles of Operation* or *ESA/390 Principles of Operation* for more information about literals.
- An equate.
- An immediate value, which is a string that represents 1 byte and can be one of the following:
  - Character (C"A")
  - Hexadecimal (X'40')
  - Binary (B'10101010')
  - Length (L'EBW000).
- An arithmetic expression, which is an expression that resolves into an arithmetic value. For example:
  - 10 + 3
  - FLD + 10

msk
is the M3 or R3 operand of an RS instruction. See *ESA/370 Principles of Operation* or *ESA/390 Principles of Operation* for more information about the RS instruction.

FILNC
processes the TPF or ALCS FILNC macro. See *TPF General Macros* or *ALCS Application Programming Reference — Assembler Language* for more information about the FILNC macro.

Note: The FILNC structure generates a WAITC internally to perform the test. Consider the effect of this test on performance when you use this form.

FINWC
processes the TPF or ALCS FINWC macro. See *TPF General Macros* or *ALCS Application Programming Reference — Assembler Language* for more information about the FINWC macro.

FIWHC
processes the TPF or ALCS FIWHC macro. See *TPF General Macros* or *ALCS Application Programming Reference — Assembler Language* for more information about the FIWHC macro.

WAITC
processes the TPF or ALCS WAITC macro. See *TPF General Macros* or *ALCS Application Programming Reference — Assembler Language* for more information about the WAITC macro.

D*num*
specifies a data level, where *num* is a data level in the range 0–F.

num
specifies a data level, where *num* is a data level in the range 0–F.

OK
evaluates the conditional expression as true if the specified TPF or ALCS macro ends successfully.

NOK
evaluates the conditional expression as true if the specified TPF or ALCS macro does not end successfully.

LEVTA
processes the TPF or ALCS LEVTA macro. See *TPF General Macros* or *ALCS Application Programming Reference — Assembler Language* for more information about the LEVTA macro.

INUSE
evaluates the conditional expression as true if the specified data level is being used.

NOTUSED
evaluates the conditional expression as true if the specified data level is available.

CPU
checks the symbolic processor ID of the ECB (field CE1CPD in the ECB).

operator
is one of the following operators:

Operator	Description
EQ	Equal
NE	Not equal
LT	Less than
LE	Less than or equal
GT	Greater than
GE	Greater than or equal.

cpuid
is the symbolic processor ID that you are checking.

DBFOUND
checks for the presence of a logical record (LREC), where:

YES
checks that the requested LREC exists.

NO
checks that the requested LREC does not exist.

See *TPFDF Programming Concepts and Reference* for more information about finding LRECs.

DBERROR
checks for serious errors, where:

YES
checks if a serious error occurred.

NO
checks if a serious error did not occur.

A *serious error* is an error that does not occur for obvious reasons and often results in a system error dump, such as an I/O error or corrupted data block.

Using this form is the same as specifying the ERROR parameter on a TPFDF assembler macro and can be used to process errors inline rather than at the end of the application program. See *TPFDF Programming Concepts and Reference* for more information about detecting errors and the SW00RTN settings for serious errors.

DBEOF
checks for an end-of-file (EOF), where:

YES
checks if an EOF condition was detected.

NO
checks if an EOF condition was not detected.

Use this form with the FULLFILE parameter on a TPFDF macro. See *TPFDF Programming Concepts and Reference* for more information about the TPFDF macros.

DBIDX
checks that an indexed detail file actually exists, where:

YES
checks if an indexed detail file exists.

NO
checks if an indexed detail file does not exist.

See *TPFDF Programming Concepts and Reference* for more information about creating indexed detail files. See *TPFDF Database Administration* for more information about indexed detail files in general.

DBEMPTY
checks for an empty subfile, where:

YES
checks if the subfile is empty.

NO
checks if the subfile is not empty.

Note: This parameter is valid only after a delete operation that does not use fullfile processing and before the next TPFDF call.

Examples of Conditional Expressions

The following sections contain examples of the different forms of conditional expressions.

Branch on Condition Code Conditional Expression

In the following example, the branch on condition code form is used for the #ELIF macro.

```
         #IF   (LTR,R0,R0,P)
           :
*   Code to process if R0 is positive
           :
         #ELIF (Z)
           :
*   Code to process if register R0 is zero
           :
         #ELSE
           :
*   Code to process if register R0 is negative
           :
         #EIF
```

Compare Conditional Expressions

The following are examples of compare conditional expressions.

- The following example shows how to compare two fields.

```
         #IF   (CLC,FLDA,NE,FLDB)
           :
*  Code to process
           :
         #EIF
```

- The following example shows how to compare two registers.

```
         #IF   (CR,R2,EQ,R3)
           :
*  Code to process
           :
         #EIF
```

- The following example shows how to compare a register to a literal.

```
         #IF   (C,R2,EQ,=F'25')
           :
*   Code to process
           :
         #EIF
```

Noncompare Conditional Expressions

The following are examples of noncompare conditional expressions.

- The following example shows how the result of an Insert Character under Mask (ICM) instruction can be used in a conditional expression.

```
         #IF   (ICM,R2,7,FLD,NZ)
           :
*  Code to process
           :
         #EIF
```

- The following example shows how to test 4 bits of a field in a conditional expression.

```
         #IF   (TM,FLD,X'F0',NO)
           :
*  Code to process
           :
         #EIF
```

TPF and ALCS Macros as Conditional Expressions

The following are examples of using the TPF and ALCS macros as conditional expressions.

- In the following example, the code is processed if the FINWC macro is processed successfully.

```
         #IF   (FINWC,D3,OK)
           :
*  Code to process
           :
         #EIF
```

- In the following example, the code is processed if the WAITC macro is ***not*** processed successfully.

```
         #IF   (WAITC,NOK)
           :
*  Code to process
           :
         #EIF
```

- In the following example, the code is processed if data level 2 (D2) is not being used.

```
         #IF   (LEVTA,D2,NOTUSED)
           :
*  Code to process
           :
         #EIF
```

Checking the CPU ID Example

The following is an example of using the CPU operand to check for the processor ID.

```
        #IF   (CPU,EQ,C'A')
          :
*   Code to process
          :
        #EIF
```

Testing SW00RTN Bits Examples

The following are examples of testing the SW00RTN bits.

- The following is an example of the DBFOUND operand.

```
          DBOPN REF=GR00SR,...
          DBRED REF=GR00SR,...
          #IF DBFOUND,YES
            :
  *   Code to process if an LREC is successfully read
            :
          #EIF
```

- The following is an example of the DBERROR operand.

```
          DBOPN REF=GR00SR,...
          DBRED REF=GR00SR,...
          #IF DBERROR,YES
            :
  *   Code to process if a serious error is found
            :
          #EIF
```

- The following is an example of the DBEOF operand.

```
          DBOPN REF=GR01SR,...
          #DO INF
             DBRED REF=GR01SR,FULLFILE,...
          #DOEX DBEOF,YES
            :
  *   Code to process if EOF condition is not detected
            :
          #EDO
```

- The following is an example of the DBIDX operand.

```
          DBOPN REF=GR01SR,...
          DBRED REF=GR01SR,...
          #IF DBIDX,NO
            :
  *   Code to process if the requested detail file does not exist
            :
          #EIF
```

Testing SW00RT2 Bits Example

The following example checks if a subfile is empty.

```
        DBOPN REF=GR00SR,...
        DBRED REF=GR00SR,...
        #IF DBEMPTY,YES
          :
*   Code to process if a subfile is not empty
          :
        #EIF
```

Condensed Forms of Conditional Expressions

Programming conventions in the TPF and ALCS environments allow the following assumptions when operands in an expression are evaluated:

- Any 2-character or 3-character operand starting with an R is a general-purpose register; for example, R3 or RGB.
- Any operand starting with the number sign (#) character is an equate value.
- Any operand starting with the characters BIT refers to a specified bit pattern for a TM type operation.

These assumptions allow you to use *condensed forms* of certain conditional expressions.

Condensed Forms of Compare

You can omit the operation code when the context of a test is not ambiguous.

Table 3 shows examples of condensed conditional expressions and the C*xx* instruction that is generated for each one.

Table 3. Instructions Generated for Condensed Forms of Compare

Conditional Expression	Generates	Remarks
`#IF  R14,EQ,R15`	CR	
`#IF  R14,GE,EBW000`	C	No slash (/) defaults to a 4-byte comparison.
`#IF  R14,LT,10(R15)`	C	
`#IF  R14,LE,=H'123'`	CH	
`#IF  R14,GT,H/24(R15)`	CH	The H/ forces a `CH`.
`#IF  EBW000,NE,FLD`	CLC	
`#IF  EBW000,LT,5`	CLI	
`#IF  EBW000,GT,C"D"`	CLI	
`#IF  EBW000,NL,#CAR`	CLI	
`#IF  EBW000,L,I/FLAG`	CLI	The I/ forces a `CLI`.

Condensed Forms of TM

For a TM instruction, you can omit the operation code (TM) if the second operand begins with the characters BIT. The second operand provides a symbolic name for the bits to be tested, as shown in the examples in Table 4.

Table 4. Instructions Generated for Condensed Forms of TM

Conditional Expression	TM Mask	Remarks
`#IF  EBW000,BIT3,OFF`	X'10'	Operand BIT*n* tests bit *n*.
`#IF  EBW000,BITS4-6,MIXED`	X'0E'	Operand BITS*n-m* tests bits *n* through *m*.
`#IF  EBW000,BITS1/3/5/7,ON`	X'55'	Operand BITS*p*/*q*/*r*/*s* tests bits *p*, *q*, *r*, and *s*.

Condensed Forms of LTR and OC

You can omit the operation code and the second operand if the first and second operands of an LTR or OC instruction are the same.

Table 5. Instructions Generated for Condensed Forms of LTR and OC

Conditional Expression	Generates
`#IF  R14,NONZERO`	LTR R14,R14
`#IF  FLD(4),ZERO`	OC FLD(4),FLD

Condensed Forms of Boolean Expressions

If the operand or condition code mnemonic (or both) are repeated in a test, you can omit the second operand or condition code mnemonic (or both).

- In the following example, the operands are the same but the condition code mnemonics are different. The statement

  ```
  #IF EBW000,GT,1,AND,EBW000,LT,5
  ```

  can be coded as:

  ```
  #IF   EBW000,GT,1,AND,LT,5
  ```

  EBW000 is implied as the operand for the AND test.
- In the following example, both the operands and the condition code mnemonics are the same. The statement

  ```
  #IF  EBW000,EQ,5,OR,EBW000,EQ,6
  ```

  can be coded as:

  ```
  #IF   EBW000,EQ,5,OR,6
  ```

  EBW000 and EQ are implied for the OR test.
- The tests and operands are determined as shown in the following example. In this example, the tests are emphasized to show each set of tests. The statement

  ```
  #IF   R14,GE,R15,AND,LE,EBW000,AND,NE,=H'123'
  ```

  generates the following tests:

  ```
  #IF   R14,GE,R15,AND,LE,EBW000,AND,NE,=H'123'

  #IF   R14,GE,R15,AND,LE,EBW000,AND,NE,=H'123'

  #IF   R14,GE,R15,AND,LE,EBW000,AND,NE,=H'123'
  ```

Processing Rules for Boolean Connectors

The ***overall processing sequence*** is as follows:

1. Processing in a group is from left to right and top to bottom.
2. Conditional expression groups are processed from left to right.

The ***sequence of evaluation in a group*** is as follows:

3. If a test in a group is true and it is followed by an OR connector, the whole group is true.
4. If a test in a group is false and it is followed by an AND connector, the whole group is false.

Note: This is not strict Boolean logic, so it is necessary to force the sequence of evaluation.

For example, the expression `A AND B OR C` is evaluated in Boolean logic as `(A AND B) OR C`.

To ensure that the expression is evaluated unambiguously, one of the following solutions is required:

a. Force two groups.

```
A AND B ORIF C
```

b. Ensure that the C is not ignored.

```
C OR A AND B
```

The ***sequence of evaluation between groups*** is as follows:

5. If a group followed by an ORIF connector is true, the group following the next ANDIF connector (if present) is checked.

   Note: This is not strict Boolean logic.
6. If a group followed by the ANDIF connector is false, the group following the next ORIF connector (if present) is checked.
7. Each ORIF connector must be preceded by either an AND connector or an ANDIF connector; otherwise, it has no grouping effect and functions as a normal OR connector.
8. Each ANDIF connector must be preceded by either an OR connector or an ORIF connector; otherwise, it has no grouping effect and functions as a normal AND.

   For example, the test:

```
A and (B or C)
```

   is coded as:

```
B OR C ANDIF A
```

   to force two groups.

Evaluating Concatenated Expressions

As a summary of the processing rules described previously, Table 6 and Table 7 on page 25 show the steps that are used to evaluate an expression based on whether the connector is AND, ANDIF, OR, or ORIF.

Table 6. Decision Table: Boolean Expression Evaluation for AND or ANDIF

Connected by AND or ANDIF?	Left Expression True?	Expression Followed by ORIF?	Result
Yes	Yes	Yes or No	Test the next conditional expression.
Yes	No	Yes	Test the expression after the next ORIF.
Yes	No	No	The whole conditional expression is FALSE.

Table 7. Decision Table: Boolean Expression Evaluation for OR or ORIF

Connected by OR or ORIF?	Left Expression True?	Expression Followed by ANDIF?	Result
Yes	Yes	Yes	Test the expression after the next ANDIF.
Yes	Yes	No	The whole conditional expression is TRUE.
Yes	No	Yes or No	Test the next conditional expression.

Boolean Expression Examples

The code that is required to check the format of an input field is a good example of where problems can occur in concatenating tests.

- The following is an example of how to check a 2-character field.

  The field called FLD contains 2 characters that can be either of the following:
  – The characters KL
  – Two numeric characters.

  This implies the following:

  ```
  (FLD='K' AND FLD+1='L') OR (FLD>='0' AND FLD<='9' AND FLD+1>='0' AND FLD+1<='9')
  ```

  There are two groups, separated by the OR connector. The ORIF connector is used to separate groups, so the expression can be written as follows:

  ```
          #IF   FLD,EQ,C'K',AND,FLD+1,EQ,C'L',
          #     ORIF,FLD,GE,C'0',AND,LE,C'9',
          #     AND,FLD+1,GE,C'0',AND,LE,C'9'     Format of field correct
            :
*   Code to process the field.
            :
          #ELSE                                   Format error
            :
*   Code to process the illegal field.
            :
          #EIF
  ```

- The following is an example of how to check a 3-character field.

  The field called FLD contains 3 characters. The first 2 characters must be KL and the third character can be either of the following:
  – The character M
  – A numeric character.

  This implies the following:

  ```
  FLD='K' AND FLD+1='L' AND (FLD+2='M' OR FLD+2>='0' AND FLD+2<='9')
  ```

  Again, there are two groups, separated by the AND connector. However, simply replacing the AND connector with the ANDIF connector does not work because of rule 8 on page 24, which states that each ANDIF connector must be preceded by either an OR or an ORIF connector; otherwise, it has no grouping effect and functions as a normal AND connector.

  The ANDIF connector must be preceded by an OR connector so the sequence has to be reversed to allow the ANDIF connector to function. The conditional expression in condensed form is:

```
         #IF   FLD+2,EQ,C'M',OR,GE,C'0',AND,LE,C'9',
         #     ANDIF,FLD,EQ,C'K',AND,
         #     FLD+1,EQ,C'L'                    Format of field correct
           :
*   Code to process the field.
           :
         #ELSE                                  Format error
           :
*   Code to process the illegal field.
           :
         #EIF
```

- The following is another example of checking a 3-character field.

  The field called FLD contains 3 characters.
  - The first character must be a K.
  - The second character can be an L or a U.
  - If the second character is a U, the third character must be either 0 or 1.

  This implies the following:

  ```
  FLD='K' AND (FLD+1='L' OR FLD+1='U' AND (FLD+2='0' OR FLD+2='1'))
  ```

  This expression uses double parentheses and, therefore, contains *nested groups*. The SPM evaluator does not support nested groups. However, the expression can be split into two groups by duplicating the first test.

  ```
  FLD='K' AND FLD+1='L' ORIF FLD='K' AND FLD+1='U' AND (FLD+2='0' OR FLD+2='1')
  ```

  This still leaves a parenthesis preceded by an AND connector, which cannot be split simply by replacing the AND connector with an ANDIF connector (see rule 5 on page 24). The expression must be rearranged as follows:

  ```
         #IF  (FLD+2,EQ,C'0'),OR
         #    (FLD+2,EQ,C'1'),ANDIF
         #    (FLD+1,EQ,C'U'),ORIF,
         #    (FLD+1,EQ,C'L'),ANDIF,
         #    (FLD,EQ,C'K')                   Format of field correct
           :
*   Code to process the field.
           :
         #ELSE                                  Format error
           :
*   Code to process the illegal field.
           :
         #EIF
  ```

TPFDF Structured Programming Macros: Reference

The following contains an alphabetic listing of the TPFDF structured programming macros (SPMs). The description of each SPM includes the following information:

Format
Provides a syntax (railroad track) diagram for the macro and a description of each parameter and variable. See “How to Read the Syntax Diagrams” on page x for more information about syntax diagrams.

Entry Requirements
Lists any special conditions that must be true when you use the macro.

Return Conditions
Lists what is returned when the macro has finished processing.

Programming Considerations
Lists any additional considerations for using the macro, including any restrictions or limitations.

Examples
Provides one or more examples that show you how to code the macro.

Related Macros
Lists where to find information about related macros.

See “TPFDF Structured Programming Macro Group Processing Diagrams” on page 79 for figures that show the processing flow of the TPFDF macro groups.

#–Line Continuation

Use this macro to continue conditional expressions across more than one line. This macro is used instead of the *continuation character* in column 72 of the previous line.

Format

inputline
is the continuation of a conditional expression and can be one of the following:

- Whole conditional expression, including those with Boolean connectors
- Whole test (symbolic assembler instructions or macro tests cannot be split).

Entry Requirements

You can only code this macro following a #DO, #IF, or #GOTO macro.

Return Conditions

Control is passed to the next sequential instruction.

Programming Considerations

- All labels used in the SPM conditional expression can be no more than 32 characters long. Any additional characters are truncated.
- You cannot use the # macro with the normal assembler continuation character.
- You must code a comma (,) immediately after the last operand on the previous line.
- When you use the #DO macro with a Boolean connector, the Boolean connector ***cannot*** be coded as the last operand on the previous line.

Examples

- In the following example, the # macro is used to continue Boolean connectors.

```
#IF   (CLI,0(R4),NL,C'0'),AND,
#     (CLI,0(R4),NH,C'9'),ORIF,
#     (CLI,0(R4),NL,C'A'),AND,
#     (CLI,0(R4),NH,C'Z')
```

- In the following example, the # macro is used to continue conditional expressions.

```
#DO   WHILE=(R2,GE,R3),
#     AND,(EBW000,BIT1,OFF),
#     UNTIL=(FIELD,ZERO)
```

Related Information

- “#DO Macro Group” on page 48
- “#GOTO Macro Group” on page 61
- “#IF Macro Group” on page 63.

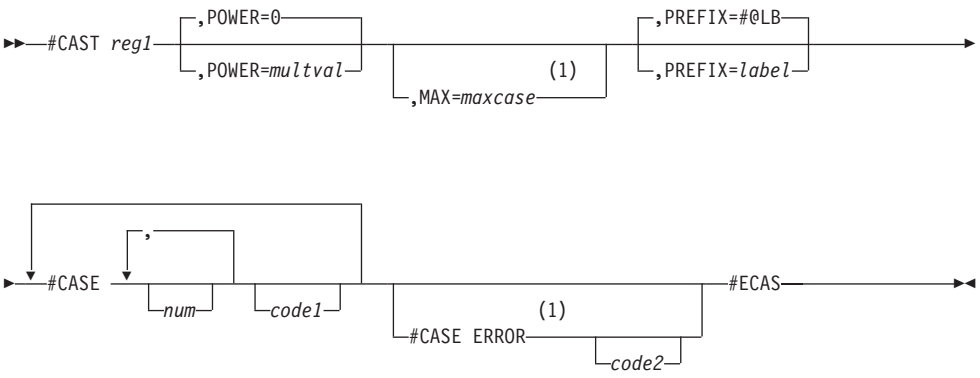
#CASE Macro Group

Use this macro group to process specific code based on a numeric value, referred to as a *case number*. The #CASE macro group includes the following macros:

- #CAST
- #CASE
- #ECAS.

See “#CASE Macro Group Processing” on page 79 for a diagram that shows the processing flow of the #CASE macro group.

Format



Notes:

1 If you specify the MAX parameter, you must also code a #CASE ERROR macro statement.

#CAST *reg1*
starts a group of #CASE selections, where *reg1* is a register that contains the case number. The case number in *reg1* is adjusted to index into a branch table generated by the #ECAS macro.

Do not use register 0 for *reg1*.

POWER=*multval*
specifies the incremental value between case numbers, where *multval* is the value. The increment value is $2^{multval}$. For example, if you specify POWER=3, the case numbers are multiples of 8. If you do not specify the POWER parameter, the case numbers are generated in increments of 1.

MAX=*maxcase*
specifies the maximum case number allowed in a group, where *maxcase* is the maximum number.

If you specify the MAX parameter, you must also code a #CASE ERROR statement.

PREFIX=*label*
specifies a prefix for all link labels that are generated by this macro group, where *label* is a 4-character name.

#CASE *num*
starts a case or group of cases, where *num* is the case number. The code up to the next #CASE macro or #ECAS macro is processed when *reg1* contains the relevant case number.

code1
is the code to process for the specified case or cases.

#CASE ERROR
specifies the start of the code to process for case numbers that are not valid. If you specify the MAX parameter on the #CAST macro, you must also code a #CASE ERROR macro statement.

code2
is the code to process for incorrect case numbers.

#ECAS
ends the selection. This macro generates a branch table entry for each case number specified in a #CASE macro statement. A dummy table entry is generated for each valid but unused case number.

Entry Requirements

None.

Return Conditions

- The value in *reg1* is modified during processing except when you specify POWER=2.
- Control is returned to the next sequential instruction after the #ECAS macro statement unless another assembler instruction or macro passes control outside the #CASE structure.

Programming Considerations

- All labels used in the SPM conditional expression can be no more than 32 characters long. Any additional characters are truncated.
- The #CAST, #CASE, and #ECAS macros can only be used with the #CASE macro group.
- Each macro statement and assembler instruction must begin on a new line in the application.
- A section of code (represented by *code1*, and so on) can consist of any number of standard assembler instructions, including other SPMs or assembler macros.
- The SPMs generate assembler instructions according to the specified macro parameters. The necessary branch instructions and link labels are generated internally to support the nonsequential processing. The SPMs generate standard link labels in the following format:

  ```
  #@LBn  EQU  *
  ```

  Where:
  #@LB is the link-label prefix.
  n is a sequence number that is generated automatically.

  You can use the PREFIX parameter to change the standard link-label prefix.
- The labels generated by macros nested inside a structure have the standard prefix #@LB unless another PREFIX parameter is specified with the inner SPMs.

- The first case number must be $2^{multval}$; *multval* can be 0. That is, the first case number can be 1, 2, 4, 8, and so on. Other case numbers in the same group must be multiples of the first case number. Use the POWER parameter to change the multiples for the case number.
- An assembly error is issued if:
  - A case number (*num*) conflicts with the POWER parameter.
  - A case number (*num*) is greater than the value specified by the MAX parameter on the #CAST macro.
- If *reg1* contains a 0, the whole case structure is bypassed and processing continues at the next instruction after the #ECAS macro.
- Use the MAX parameter on the #CAST macro and the #CASE ERROR macro statement to check for case numbers, specified by *reg1*, that are not valid. During processing, a case number is not valid if the value of *reg1* is:
  - Greater than the largest case number specified by the MAX parameter on the #CAST macro
  - Not a multiple of the first case number
  - A negative number.
- If you do not specify the MAX parameter:
  - All case numbers are treated as valid during assembly.
  - No checking is performed during processing.
  - A case number greater than the largest specified in any of the #CASE macro statements causes an unpredictable branch in the program.
- All valid, unspecified case numbers bypass the structure without performing any cases, and processing continues at the next instruction after the #ECAS macro.
- Only one section of code is performed each time the #CASE structure is processed.

Examples

- In the following example, the case numbers are multiples of 8 (2^3). The #ECAS macro generates a 4-word branch table with entries as follows:

  8 Branch to code for cases 8 and 32

  16 Branch to code for case 16

  24 Dummy entry (case not used); branch to the next sequential instruction after the #ECAS macro

  32 Branch to code for cases 8 and 32.

  A value other than 8, 16, 24, or 32 in R2 will cause results that cannot be predicted.

```
         #CAST R2,POWER=3
         #CASE 32,8
          :
*   Code to process if R2 contains 8 or 32
          :
         #CASE 16
          :
*   Code to process if R2 contains 16
          :
         #ECAS
```

- In the following example, the maximum number of cases allowed is 10. The case number increment defaults to 1. The #ECAS macro generates a 10-word branch table with entries as follows:

1 Branch to code for cases 1, 3, 5, and 7
2 Dummy entry
3 Branch to code for cases 1, 3, 5, and 7
4 Branch to code for cases 4 and 8
5 Branch to code for cases 1, 3, 5, and 7
6 Dummy entry
7 Branch to code for cases 1, 3, 5, and 7
8 Branch to code for cases 4 and 8
9 Dummy entry
10 Dummy entry

Case numbers 2, 6, 9, and 10 bypass the structure, and processing continues with the instruction after the #ECAS macro.

```
         #CAST R2,MAX=10
         #CASE 1,3,5,7
          :
*   Code to process if R2 contains 1, 3, 5, or 7
          :
         #CASE 4,8
          :
*   Code to process if R2 contains 4 or 8
          :
         #CASE ERROR
          :
*   Code to process if R2 contains a negative value or a value greater
    than 10.
          :
         #ECAS
```

Related Information

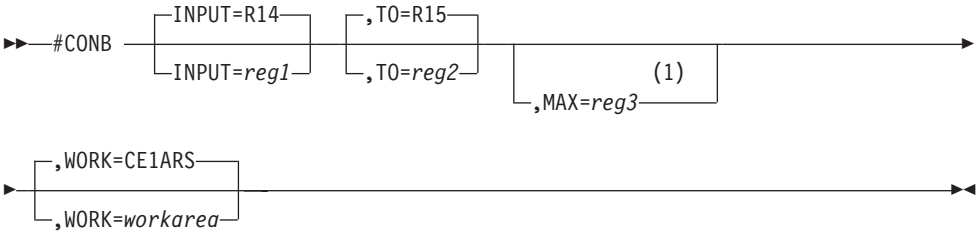
None.

#CONB–Convert Character Decimal to Binary

Use this macro to generate inline code to convert a *character decimal* number to a binary value. A character decimal number is a number represented in a string that contains only EBCDIC 0–9.

Note: See Table 1 on page 10 for a summary of all the conversion macros.

Format



Notes:

1 If you do not specify the MAX parameter, the maximum number of characters defaults to 16 bytes.

INPUT=*reg1*
specifies a register, *reg1*, that points to the start of the string to convert.

TO=*reg2*
specifies a register, *reg2*, that will contain the converted binary value. Do ***not*** use R0 for *reg2*.

MAX=*reg3*
specifies a register, *reg3*, that contains a binary number indicating the maximum number of characters to convert. Leading zeros (C"0") in the input string are ignored. You can specify a maximum of 16 bytes.

WORK=*workarea*
specifies a 16-byte work area.

Entry Requirements

None.

Return Conditions

- *reg1* points to the next byte following the last byte that was converted.
- *reg2* contains one of the following:
  - The resulting binary value.
  - X'FFFFFFFF', if the first character is not numeric; for example, the string C"A94732".
  - X'FFFFFFFF', if more than 9 digits following any leading zeros are specified; for example, the string C"1234567890".
- The contents of *reg3* are overwritten during the conversion process.

Programming Considerations

- You can specify the parameters for this macro in any order.
- All labels used in the SPM conditional expression can be no more than 32 characters long. Any additional characters are truncated.
- This macro converts the input string only up to the first nondigit character. For example, if the input string contains C"789B2", the macro converts only C"789", and *reg1* points to B.

Examples

- In the following example:

Before the conversion: R14 points to CONB0, which contains C"12713971".
R15 contents are unknown.
R0 contains 8.
After the conversion: R14 points to CONB0+8.
R15 contains X'C1FFF3'.
R0 contents are overwritten.

```
          LA    R14,CONB0                    SET UP INPUT ADDRESS
          LA    R0,8                         SET UP INPUT LENGTH
          #CONB INPUT=R14,TO=R15,MAX=R0      EBCDIC-STRING TO BINARY
           :
CONB0     DC    C'12713971'                  DECIMAL INPUT STRING
```

- In the following example:

Before the conversion: R14 points to CONB4, which contains C"789B2".
R15 contents are unknown.
R0 contains 16.
After the conversion: R14 points to CONB4+3.
R15 contains X'00000315'.
R0 contents are overwritten.

```
          LA    R14,CONB4                    SET UP INPUT ADDRESS
          LA    R0,16                        SET UP INPUT LENGTH
          #CONB INPUT=R14,TO=R15,WORK=EBX024,MAX=R0
           :
CONB4     DC    C'789B2'                     DECIMAL INPUT STRING
```

Related Information

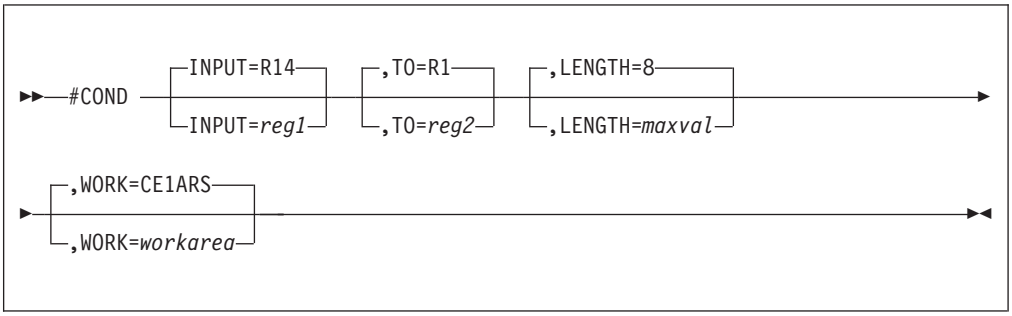
"#CONH–Convert Character Hexadecimal to Binary" on page 37.

#COND–Convert Binary to Character Decimal

Use this macro to generate inline code to convert a binary number to character decimal with leading zeros.

Note: See Table 1 on page 10 for a summary of all the conversion macros.

Format



INPUT=*reg1*
specifies a register, *reg1*, that contains the binary number to convert.

TO=*reg2*
specifies a register, *reg2*, that points to the location that will contain the converted value.

LENGTH=*maxval*
specifies the length of the resulting character decimal string, where *maxval* is the length. The maximum value for *maxval* is 15.

WORK=*workarea*
specifies an 8-byte work area.

Entry Requirements

None.

Return Conditions

- *reg2* points to the next available byte following the result.
- If the length specified is too short, the result is truncated to the left. If the length specified is longer then the input string, the result is padded to the left with zeros (X'F0').

Programming Considerations

- You can specify the parameters for this macro in any order.
- All labels used in the SPM conditional expression can be no more than 32 characters long. Any additional characters are truncated.

Examples

- In the following example:

```
Before the conversion:   R14 contains X'0000F394'.
                         R15 points to EBW000.
                         The length is 8.
```

After the conversion: R14 is unchanged.
EBW000 contains C"00062356".
R15 points to EBW000+8.

```
          L     R14,COND1                    SET UP BINARY VALUE
          LA    R15,EBW000                   WHERE TO PLACE
          #COND INPUT=R14,TO=R15,LENGTH=8    BINARY TO EBCDIC
          :
COND1     DC    X'0000F394'                  BINARY VALUE (DEC 62356)
```

- In the following example:

Before the conversion: R14 contains X'0000F394'.
R15 points to EBW020.
The length is 5.
After the conversion: R14 is unchanged.
EBW020 contains C"62356".
R15 points to EBW020+5.

```
          L     R14,COND2                    SET UP BINARY VALUE
          LA    R15,EBW020                   WHERE TO PLACE
          #COND INPUT=R14,TO=R15,LENGTH=5    BINARY TO EBCDIC
           :
COND2     DC    X'0000F394'                  BINARY VALUE (DEC 62356)
```

- In the following example:

Before the conversion: R14 contains F"355".
R15 points to EBW040.
The length is 2.
After the conversion: R14 is unchanged.
EBW040 contains C"55".
R15 points to EBW040+2.

```
          L     R14,COND3                    SET UP BINARY VALUE
          LA    R15,EBW040                   WHERE TO PLACE
          #COND INPUT=R14,LENGTH=2,TO=R15    BINARY TO EBCDIC
           :
COND3     DC    F"355"                       BINARY VALUE
```

Related Information

"#CONS–Convert Binary to Character Decimal with Zero Suppression" on page 42.

#CONH–Convert Character Hexadecimal to Binary

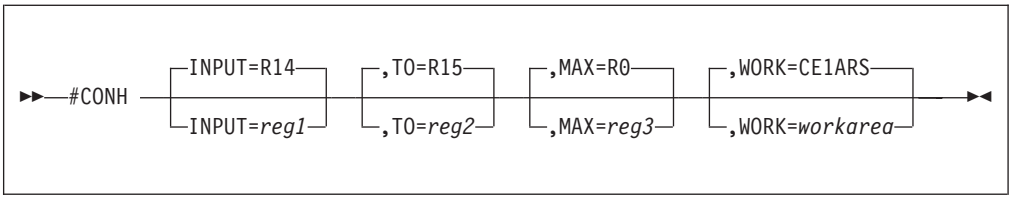
Use this macro to generate inline code to convert a character hexadecimal number to a binary value. Each pair of character hexadecimal digits converts to 1 byte of binary.

This macro only converts data that corresponds to the EBCDIC characters A–Z and 0–9. The corresponding hexadecimal values are as follows:

Hexadecimal Value	EBCDIC Value
X'C1'–X'C9'	A–I
X'D1'–X'D9'	J–R
X'E2'–X'E9'	S–Z
X'F0'–X'F9'	0–9

Note: See Table 1 on page 10 for a summary of all the conversion macros.

Format



INPUT=*reg1*
specifies a register, *reg1*, that points to the start of the string to convert.

TO=*reg2*
specifies a register, *reg2*, that points to the location that will contain the converted value.

MAX=*reg3*
specifies a register, *reg3*, that contains the maximum number of characters to convert. Specify this value as a multiple of 2. If the value is ***not*** a multiple of 2, it will be rounded up to a multiple of 2. For example, if you specify 3, the value used will be 4.

WORK=*workarea*
specifies an 8-byte work area.

Entry Requirements

None.

Return Conditions

- *reg1* points to the next byte immediately following the input string. The input string is overwritten during the conversion process.
- *reg2* points to the next available byte following the output string.
- *reg3* contains the number of bytes remaining to be converted (normally zero). If the value specified for *reg3* was not a multiple of 2, *reg3* contains X'FFFFFFFF'.

Programming Considerations

- You can specify the parameters for this macro in any order.

- All labels used in the SPM conditional expression can be no more than 32 characters long. Any additional characters are truncated.
- This macro converts the input string only up to the first character that is not A–Z or 0–9. For example, if the input string contains C"7B9*2", the macro converts only C"7B9", and *reg1* points to the asterisk (*).

Examples

- In the following example:

Before the conversion: R14 points to EBX000, which contains C"12713971".
R15 points to EBW000.
R0 contains 8.
After the conversion: R14 points to EBX000+8.
EBW000 contains X'12713971'.
R15 points to EBW000+4.
R0 is zero (no bytes left to convert).

```
MVC   EBX000(8),=C'12713971'
LA    R14,EBX000
LA    R15,EBW000
LA    R0,8
#CONH INPUT=R14,TO=R15,MAX=R0
```

- In the following example:

Before the conversion: R14 points to EBX000, which contains C"ACD9E7".
R15 points to EBW010.
R0 contains 6.
After the conversion: R14 points to EBX000+6.
EBW010 contains X'ACD9E7'.
R15 points to EBW010+3.
R0 is zero (no bytes left to convert).

```
MVC   EBX000(6),=C'ACD9E7'          SET UP INPUT STRING
LA    R14,EBX000                    SET UP INPUT ADDRESS
LA    R15,EBW010                    WHERE TO PLACE
LA    R0,6                          NUMBER OF INPUT CHARS
#CONH INPUT=R14,TO=R15,MAX=R0       HEXADECIMAL TO BINARY
```

- In the following example:

Before the conversion: R14 points to EBX020, which contains C"AB35*CDE".
R15 points to EBW030.
R0 contains 8.
After the conversion: R14 points to EBX020+4 (C"*").
EBW030 contains X'AB35'.
R15 points to EBW030+2.
R0 contains 4.

```
MVC   EBX020(8),=C'AB35*CDE'        SET UP INPUT STRING
LA    R14,EBX020                    SET UP INPUT ADDRESS
LA    R15,EBW030                    WHERE TO PLACE
LA    R0,8                          NUMBER OF INPUT CHARS
#CONH INPUT=R14,TO=R15,MAX=R0       HEXADECIMAL TO BINARY
```

Related Information

"#CONB–Convert Character Decimal to Binary" on page 33.

#CONP–Convert Binary to Character Hexadecimal with EBCDIC Interpretation

Use this macro to generate inline code to convert binary data to *character hexadecimal*. A character hexadecimal number is a hexadecimal number represented in a string that contains only EBCDIC 0–9 and A–F.

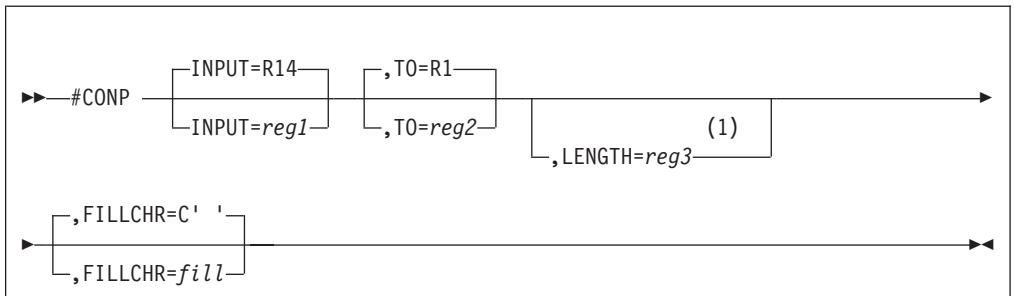
The data is converted to character hexadecimal unless it is a binary value corresponding to the EBCDIC characters A–Z and 0–9. The corresponding hexadecimal values are as follows:

Hexadecimal Value	EBCDIC Value
X'C1'–X'C9'	A–I
X'D1'–X'D9'	J–R
X'E2'–X'E9'	S–Z
X'F0'–X'F9'	0–9

A fill character is inserted in the output string to differentiate between these characters and the hexadecimal characters (see the FILLCHR parameter).

Note: See Table 1 on page 10 for a summary of all the conversion macros.

Format



Notes:

1 If you do not specify the LENGTH parameter, the length defaults to 1 character.

INPUT=*reg1*
specifies a register, *reg1*, that points to the start of the string to convert.

TO=*reg2*
specifies a register, *reg2*, that points to the location that will contain the converted value.

LENGTH=*reg3*
specifies a register, *reg3*, that contains the number of characters to convert.

FILLCHR=*fill*
specifies the character that pads the converted output, where *fill* is the character specified as an immediate value.

For example, FILLCHR=C"." specifies a "." for the fill character. The string X'C1C2FFC4' is converted to C".A.BFF.D".

Entry Requirements

None.

Return Conditions

- *reg1* points to the next byte immediately following the input string.
- *reg2* points to the next available byte following the output string.
- *reg3* contains X'00000000'.

Programming Considerations

- You can specify the parameters for this macro in any order.
- All labels used in the SPM conditional expression can be no more than 32 characters long. Any additional characters are truncated.

Examples

- In the following example:

Before the conversion: R14 points to CONP0, which contains X'C1FFF3'.
R15 points to EBW000.
R0 (the length) contains 3.
After the conversion: R14 points to CONP0+3.
EBW000 contains C" AFF 3".
R15 points to EBW000+6.
R0 contains 0.

```
          LA    R14,CONP0
          LA    R15,EBW000
          LA    R0,L'CONP0
          #CONP INPUT=R14,TO=R15,LENGTH=R0,FILLCHR=C' '
           :
CONP0     DC    X'C1FFF3'
```

- In the following example:

Before the conversion: R14 points to CONP1, which contains X'C1C2FFC4'.
R15 points to EBW050.
The length defaults to 1 byte.
After the conversion: R14 points to CONP1+1.
EBW050 contains C" A".
R15 points to EBW050+2.

```
          LA    R14,CONP1
          LA    R15,EBW050
          #CONP INPUT=14,TO=R15,FILLCHR=C' '
           :
CONP1     DC    X'C1C2FFC4'
```

- In the following example:

Before the conversion: R14 points to CONP2, which contains X'C3F361F5C4C1'.
R15 points to EBW070.
R0 (the length) contains 5.
After the conversion: R14 points to CONP2+5.
EBW070 contains C"/C/361/5/D".
R15 points to EBW070+10.
R0 contains 0.

```
         LA    R14,CONP2
         LA    R15,EBW070
         LA    R0,5
         #CONP INPUT=R14,TO=R15,LENGTH=R0,FILLCHR=C"/"
          :
CONP2    DC    X'C3F361F5C4C1'
```

Related Information

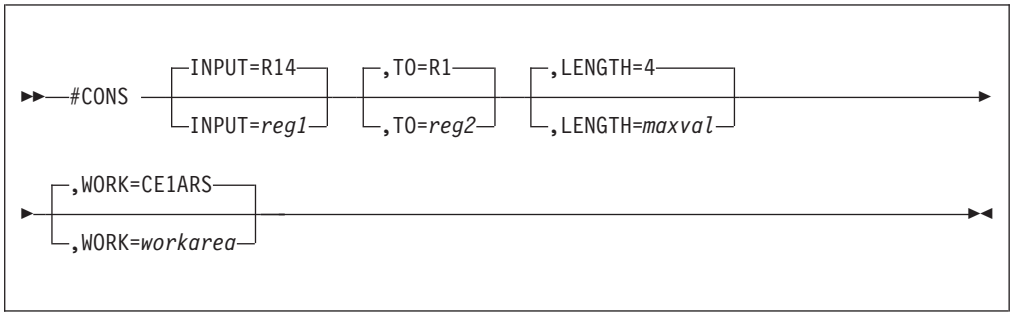
"#CONX–Convert Binary to Character Hexadecimal" on page 46.

#CONS–Convert Binary to Character Decimal with Zero Suppression

Use this macro to generate inline code to convert a binary number to character decimal with leading blanks.

Note: See Table 1 on page 10 for a summary of all the conversion macros.

Format



INPUT=*reg1*
specifies a register, *reg1*, that contains the binary number to be converted.

TO=*reg2*
specifies a register, *reg2*, that points to the location that will contain the converted value.

LENGTH=*maxval*
specifies the length of the resulting character decimal string, where *maxval* is the length.

You can specify a maximum value of 15 for the MAX parameter.

WORK=*workarea*
specifies an 8-byte work area.

Entry Requirements

None.

Return Conditions

- The contents of *reg1* are overwritten during the conversion process.
- *reg2* points to the next available byte following the output string.
- If the length specified is too short, the result is truncated to the left. If the length specified is longer than the input string, the result is padded to the left with blanks (X'40').

Programming Considerations

- You can specify the parameters for this macro in any order.
- All labels used in the SPM conditional expression can be no more than 32 characters long. Any additional characters are truncated.

Examples

- In the following example,

Before the conversion: R14 contains X'0000F394'.
R15 points to EBW000.
The length is 8.
After the conversion: R14 is overwritten.
EBW000 contains C" 62356".
R15 points to EBW000+8.

```
          L     R14,CONS1
          LA    R15,EBW000
          #CONS INPUT=R14,TO=R15,LENGTH=8
           :
CONS1     DC    X'0000F394'
```

- In the following example,

Before the conversion: R14 contains X'0000F394'.
R15 points to EBW080.
The length is 4.
After the conversion: R14 is overwritten.
EBW080 contains C"2356".
R15 points to EBW080+4.

```
          L     R14,CONS2
          LA    R15,EBW080
          #CONS INPUT=R14,TO=R15,LENGTH=4
           :
CONS2     DC    X'0000F394'
```

- In the following example,

Before the conversion: R14 contains F"355".
R15 points to EBW040.
The length is 7.
After the conversion: R14 is overwritten.
EBW040 contains C" 355".
R15 points to EBW040+7.

```
          L     R14,CONS3
          LA    R15,EBW040
          #CONS INPUT=R14,TO=R15,LENGTH=7
           :
CONS3     DC    F"355"
```

Related Information

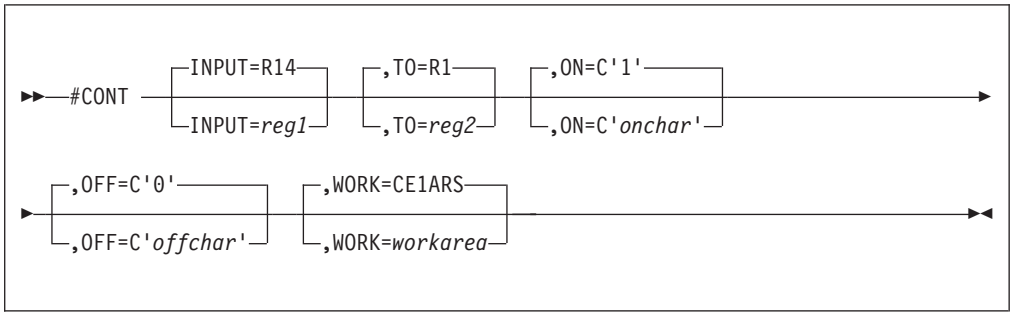
"#COND–Convert Binary to Character Decimal" on page 35.

#CONT–Convert Binary to Character Binary

Use this macro to generate inline code to convert a binary number in the low-order byte of a register to *character binary*. A character binary number is a binary number represented in a string that contains only EBCDIC 0 or 1.

Note: See Table 1 on page 10 for a summary of all the conversion macros.

Format



INPUT=*reg1*
specifies a register, *reg1*, that contains the binary number to convert.

TO=*reg2*
specifies a register, *reg2*, that points to the location that will contain the converted value.

ON=C'*onchar***'**
specifies the character used to represent binary 1 in the output string, where *onchar* is the character.

OFF=C'*offchar***'**
specifies the character used to represent binary 0 in the output string, where *offchar* is the character.

WORK=*workarea*
specifies a 5-byte work area.

Entry Requirements

None.

Return Conditions

- The contents of *reg1* are overwritten during the conversion process.
- *reg2* points to the next available byte following the output string.

Programming Considerations

- You can specify the parameters for this macro in any order.
- All labels used in the SPM conditional expression can be no more than 32 characters long. Any additional characters are truncated.

Examples

- In the following example,

Before the conversion: R14 contains F"43".
R15 points to EBW000.
ON is 1 and OFF is 0.
After the conversion: R14 is overwritten.
EBW000 contains C"00101011".
R15 points to EBW000+8.

```
          L     R14,CONT0                    SET UP BINARY VALUE
          LA    R15,EBW000                   WHERE TO PLACE
          #CONT INPUT=R14,TO=R15,ON='1',OFF=C'0',WORK=CE1ARS
           :
CONT0     DC    F'43'                        BINARY VALUE (X'2B')
```

- In the following example,

Before the conversion: R14 contains X'ABF1FF23'.
R15 points to EBW030.
ON is 1 and OFF is 0.
After the conversion: R14 is overwritten.
EBW030 contains C"00100011".
R15 points to EBW030+8.

```
          L     R14,CONT1                    SET UP BINARY VALUE
          LA    R15,EBW030                   WHERE TO PLACE
          #CONT INPUT=R14,TO=R15              LOW-ORDER BYTE TO BINARY
           :
CONT1     DC    X'ABF1FF23'                  BINARY VALUE
```

- In the following example:

Before the conversion: R14 contains F"241".
R15 points to EBW010.
ON is "*" and OFF is "-".
After the conversion: R14 is overwritten.
EBW010 contains C"****---*".
R15 points to EBW010+8.

```
          L     R4,CONT2                     SET UP BINARY VALUE
          LA    R15,EBW010                   WHERE TO PLACE
          #CONT INPUT=R14,TO=R15,ON=C"*",OFF=C"-"
    LOW-ORDER BYTE TO BINARY          : CONT2    DC    F'241'  BINARY VALUE
(X'F1')
```

Related Information

None.

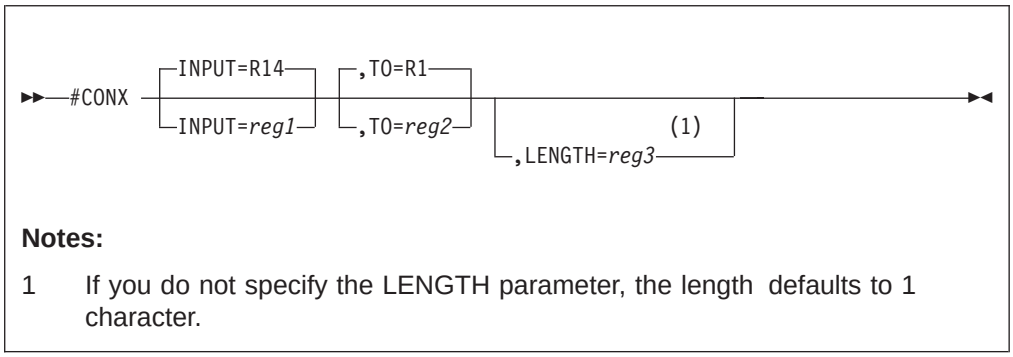
#CONX–Convert Binary to Character Hexadecimal

Use this macro to generate inline code to convert binary data to character hexadecimal. Unlike the #CONP macro, #CONX converts each byte to character hexadecimal. For example:

- #CONX converts X'C1' to C"C1" (X'C3F1').
- #CONP converts X'C1' to C" A" (X'40C1').

Note: See Table 1 on page 10 for a summary of all the conversion macros.

Format



Notes:

1 If you do not specify the LENGTH parameter, the length defaults to 1 character.

INPUT=*reg1*
specifies a register, *reg1*, that points to the start of the string to convert.

TO=*reg2*
specifies a register, *reg2*, that points to the location that will contain the converted value.

LENGTH=*reg3*
specifies a register, *reg3*, that contains the number of bytes to convert.

Entry Requirements

None.

Return Conditions

- *reg1* points to the next byte immediately following the input string.
- *reg2* points to the next available byte, which contains a blank.
- *reg3* contains X'00000000'.

Programming Considerations

- You can specify the parameters for this macro in any order.
- All labels used in the SPM conditional expression can be no more than 32 characters long. Any additional characters are truncated.

Examples

- In the following example,

Before the conversion: R14 points to CONX0, which contains X'C1FFF3'.
R15 points to EBW000.
Length is 3.

After the conversion: R14 points to CONX0+3.
EBW000 contains C"C1FFF3".
R15 points to EBW000+6.

```
          LA    R14,CONX0
          LA    R15,EBW000
          LA    R0,L'CONX0
          #CONX INPUT=R14,TO=R15,LENGTH=R0
           :
CONX0     DC    X'C1FFF3'
```

- In the following example,

Before the conversion: R14 points to CONX1, which contains X'C1C2FFC4'.
R15 points to EBW050.
The length defaults to 1.
After the conversion: R14 points to CONX1+1.
EBW050 contains C"C1".
R15 points to EBW050+2.

```
          LA    R14,CONX1
          LA    R15,EBW050
          #CONX INPUT=R14,TO=R15
           :
CONX1     DC    X'C1C2FFC4'
```

- In the following example,

Before the conversion: R14 points to CONX2, which contains X'C3F361F5C4C1'.
R15 points to EBW070.
Length is 5.
After the conversion: R14 points to CONX2+5.
EBW070 contains C"C3F361F5C4".
R15 points to EBW070+10.

```
          LA    R14,CONX2
          LA    R15,EBW070
          LA    R0,5
          #CONX INPUT=R14,TO=R15,LENGTH=R0
           :
CONX2     DC    X'C3F361F5C4C1'
```

Related Information

"#CONP–Convert Binary to Character Hexadecimal with EBCDIC Interpretation" on page 39.

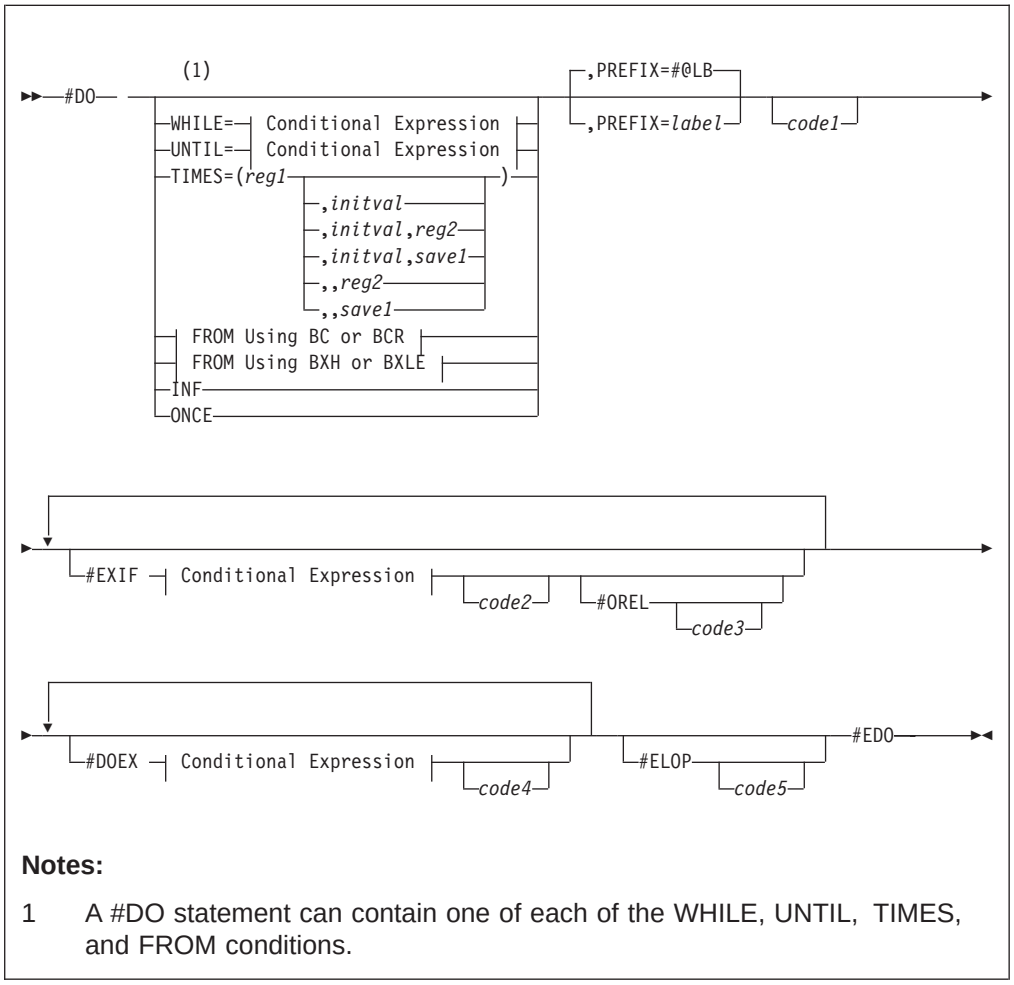
#DO Macro Group

Use this macro group to process specific code based on one of the following:
- A conditional expression
- A branch register (BCT, BCTR, BXLE, or BXH).

The #DO macro group includes the following macros:
- #EXIF
- #OREL
- #DOEX
- #ELOP
- #EDO.

See "#DO Macro Group Processing" on page 80 for diagrams that show the processing flow of the #DO macro group.

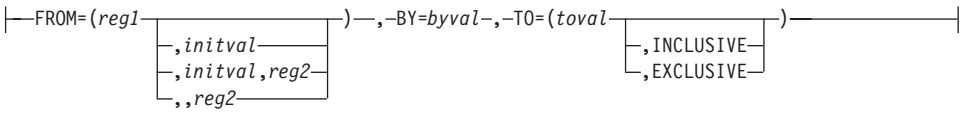
Format



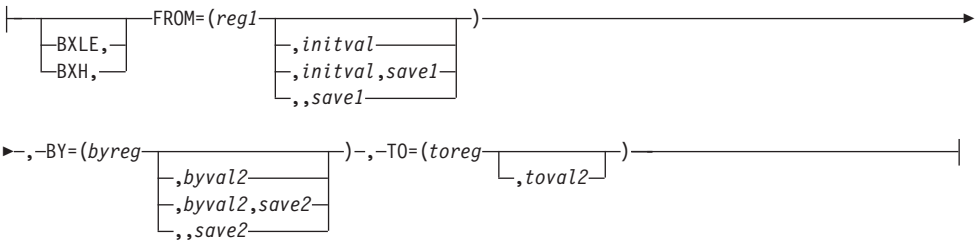
Notes:

1 A #DO statement can contain one of each of the WHILE, UNTIL, TIMES, and FROM conditions.

FROM Using BC or BCR:



FROM Using BXH or BXLE:



#DO
specifies the start of the #DO structure.

WHILE
specifies the start of a #DO WHILE loop based on a conditional expression. Any code that follows this parameter is processed only if and while the conditional expression is true. See “Conditional Expression Format” on page 14 for information about the syntax of a conditional expression.

UNTIL
specifies the start of a #DO UNTIL loop based on a conditional expression. Any code that follows this parameter is processed at least once and is repeated until the conditional expression is true. See “Conditional Expression Format” on page 14 for information about the syntax of a conditional expression.

TIMES
specifies the number of times to repeat the loop. Any code that follows this parameter is repeated based on a loop counter.

FROM
specifies the number of times to repeat the loop. Any code that follows this parameter is repeated based on a loop counter.

The form of #DO FROM using a BC or BCR loop is the basic form. The form of #DO FROM using a BXH or BXLE loop allows values to be saved during the processing of the loop. The type of loop generated depends on the following:

- Number of parameters
- Use of initial values
- Use of negative values
- Number of registers used.

Note: #DO FROM is only available for compatibility with older applications. Use the #DO TIMES form for new applications.

INF
specifies an unconditional loop. You must code a #DOEX or #EXIF macro to avoid creating an infinite loop.

ONCE
creates a *non-loop* where the processing is performed only once. This

parameter allows you to use the conditional branching macros (#EXIF, #DOEX, and so on) to construct a procedure that is processed the same way as inline code.

reg1
is a register that contains the loop counter. When you specify the TIMES parameter, the value in *reg1* must be positive.

initval
is an initial value for the loop counter that gets loaded into the register specified by *reg1*.

The value of *initval* can be:
- A numeric value
- Another general register (enclosed in parentheses)
- A literal (byte, halfword, or fullword)
- A length (with a value less than 4096)
- An equate
- A label
- An address.

You can enter an address in one of the following ways:
- A/*label*
- L/*label*

In both cases, the initial value is the address of label *label*.

You can enter labels in one of the following ways:
- X/*label*, which indicates that the initial value is the 1 byte at *label*.
- H/*label*, which indicates that the initial value is the 2 bytes (halfword) starting at *label*.
- *label*, which indicates that the initial value is the 4 bytes (fullword) starting at *label*.

reg2
is the branch register. If you specify *reg2* with the TIMES parameter, a BCTR loop is generated. If you do not specify *reg2* with the TIMES parameter, a BCT loop is generated. If you specify *reg2* with the FROM parameter, a BCR loop is generated. If you do not specify *reg2* with the FROM parameter, a BC loop is generated.

save1
is a fullword area where *reg1* is saved. This allows *reg1* to be used during the execution of the loop. If you specify *save1* with the TIMES parameter, a BCT loop is generated.

BY=*byval*
specifies the value with which to increment *reg1*, where *byval* is the value. This value is added to the value in *reg1* at the end of each iteration.

The valid values for *byval* are the same as for *initval*, with the exception of single-byte fields or single-byte literals.

TO=*toval*
specifies when to end the loop, where *toval* is a field that contains the ending value for the loop. The value in *toval* is compared with the value in *reg1* at the end of each iteration.

The valid values for *toval* are the same as for *initval*, with the exception of single-byte fields or single-byte literals.

INCLUSIVE
processes the loop including the ending value, *toval*.

EXCLUSIVE
processes the loop excluding the ending value, *toval*.

BXLE
specifies a BXLE loop.

BXH
specifies a BXH loop.

BY=*byreg*
specifies the increment value, where *byreg* is a register that contains the value with which to increment *reg1*.

byval2
is an initial increment value to load into the register specified by *byreg*.

save2
is a full or double word save area into which *byreg* and *toreg* are saved. They are stored at the beginning of each iteration and reloaded at the end of each iteration before the BXLE or BXH loop. Only 1 fullword is required during the loop processing if *byreg* and *toreg* are the same odd-numbered register.

TO=*toreg*
specifies when to end the loop, where *toreg* is a register that contains the ending value for the loop, where:

- If *byreg* is an even-numbered register, *toreg* must be the next (odd-numbered) register.
- If *byreg* is an odd-numbered register, *toreg* must be the same register.

toval2
is an initial value to load in *toreg*.

PREFIX=*label*
specifies a prefix for all link labels generated by this macro group, where *label* is a 4-character alphabetic name.

code1
is the code to process.

#EXIF
specifies the start of the exit code to process when the conditional expression is true. See “Conditional Expression Format” on page 14 for information about the syntax of a conditional expression. After the exit code is performed, processing exits from the whole #DO group. If the conditional expression is false, #EXIF branches to the next #OREL macro.

code2
is the exit code to process when the #EXIF condition is true. The exit code consists of any code between the #EXIF macro and the next sequential #OREL macro.

#OREL
specifies the start of the code to process when the previous #EXIF condition is not true. After the code is processed, control returns to the top of the loop and processing continues.

For clarity, it is recommended that you always code a matching #OREL macro for each #EXIF macro. However, if you do not specify the #OREL macro, it will be automatically generated between a #EXIF macro and any of the following:

- #DOEX macro
- #ELOP macro
- Another #EXIF macro.

code3
is the code to process when the previous #EXIF condition is false.

#DOEX
exits from the loop based on a conditional expression. See “Conditional Expression Format” on page 14 for information about the syntax of a conditional expression. If the conditional expression is true, #DOEX branches to the #ELOP macro. If the #ELOP macro is not specified, processing branches to the #EDO macro.

Note: The #DOEX macro provides a conditional exit from the iteration loop. This optional exit is a violation of structured programming rules. Restrict the use of this macro to special cases where you would otherwise have to:

- Use a #GOTO macro. See “#GOTO Macro Group” on page 61 for more information about the #GOTO macro.
- Set and test an additional indicator.

code4
is the code to process when the #DOEX condition is false.

#ELOP
ends the iteration loop and specifies the start of any code to be processed at the end of the loop. The #ELOP macro must be the last structured programming macro (SPM) in the #DO group before the #EDO macro. If you do not specify the #ELOP macro, it is automatically generated by the #EDO macro.

code5
is the code to process at the end of the loop. This code is processed only once when the iteration loop ends. This code is bypassed only when the iteration is ended by a true #EXIF condition.

#EDO
specifies the end of the loop-end processing and the whole #DO group.

Entry Requirements

None.

Return Conditions

Control is returned to the next sequential instruction after the #EDO macro statement unless another assembler instruction or macro passes control outside the #DO structure.

Programming Considerations

- All labels used in the SPM conditional expression can be no more than 32 characters long. Any additional characters are truncated.
- The #DO, #EXIF, #OREL, #DOEX, #ELOP, and #EDO macros can only be used with the #DO macro group.
- Each macro statement and assembler instruction must begin on a new line in the application.
- A section of code (represented by *code1*, and so on) can consist of any number of standard assembler instructions, including other SPMs or assembler macros.

- The SPMs generate assembler instructions according to the specified macro parameters. The necessary branch instructions and link labels are generated internally to support the nonsequential processing. The SPMs generate standard link labels in the following format:

  ```
  #@LBn  EQU  *
  ```

  Where:
  #@LB is the link-label prefix.
  n is a sequence number that is generated automatically.

  You can use the PREFIX parameter to change the standard link-label prefix.
- The labels generated by macros nested inside a structure have the standard prefix #@LB unless another PREFIX parameter is specified with the inner SPMs.
- If symbols referred to by *reg1*, *reg2*, *byval*, *toval*, *byreg*, or *toreg* are changed during loop processing, the results cannot be predicted.
- For #DO FROM using a BXH or BXLE loop, if all the parameter registers are loaded previously with the values necessary to process the loop, you must specify the BXH or BXLE parameter. See *ESA/370 Principles of Operation* or *ESA/390 Principles of Operation* for more information about the BXH and BXLE instructions.
- A #DO macro statement can contain one of each of the following parameters:
  – WHILE
  – UNTIL
  – TIMES
  – FROM.
- Additional loop-end processing can be performed (only once) between the #ELOP and #EDO macros.
- The #ELOP macro is reached when:
  – The iteration ends because the #DO WHILE condition is false (normal loop completion)
  – The iteration ends because the #DO UNTIL condition is true (normal loop completion)
  – The iteration ends with a #DOEX condition.
- You can also force an exit with a #EXIF macro combined with a #OREL macro. If the #EXIF condition is true, any code between the #EXIF macro and the next #OREL macro is processed and exits to the #EDO macro. If the #EXIF condition is false, processing continues after the #OREL macro.

Examples

- In the following #DO WHILE example, a #DOEX macro is used to exit the loop if R2 = R3.

```
         #DO   WHILE=(OC,FLD,FLD,NZ)
           :
*   Code to be processed repeatedly only if FLD is not zero.
*   This code will be done more than once only as long as
*   R2 does not equal R3 because of the following #DOEX.
           :
         #DOEX (CR,R2,EQ,R3)
           :
*   Exit the loop if R2 = R3.
*   Code to be processed repeatedly if FLD is not zero and
*   R2 does not equal R3.
           :
         #EDO
```

- The following is an example of a simple #DO UNTIL loop.

```
         #DO   UNTIL=(CLC,FLDA,GE,FLDB)
          :
*   Code to be processed at least once, and more than
*   once as long as FLDA remains less than FLDB.
          :
         #EDO
```

- The following is an example of a full #DO UNTIL loop using all the #DO group parameters.

```
         #DO   UNTIL=(CR,R2,GE,R3)
           :
*   Code to process until R2 >= R3.
*   This is done on each iteration, and at least once.
           :
         #EXIF TM,0(R2),X'FF',O
           :
*   Code to process if the bits are set to X'FF'.
*   Exit processing is performed (once only) the first
*   time the TM results in ones.
*   Processing continues at the #EDO macro.
           :
         #OREL
           :
*   Code to process if the bits are not set.
*   Processing to be done on each iteration, as long as no exits
*   are taken.
           :
         #EXIF CH,R4,NL,MAX
           :
*   Code for another #EXIF process: Maximum reached.
*   Exit processing to be performed (once only) the first
*   time R4 >= MAX.
*   Processing continues at the #EDO macro.
           :
*** #OREL generated automatically here
*
         #DOEX CLC,4(4,R2),EQ,=F'0'
           :
*   If #DOEX indicates no more items, go to #ELOP macro for
*   loop-end processing.
*
*   Code to be processed on each iteration, as long as no exits
*   are taken.
           :
         #ELOP
           :
*   End of loop, start loop-end processing.
*   Loop-end processing is done once only as a result of
*   iteration termination by the UNTIL condition (R2 >= R3),
*   or because the #DOEX exit was taken.
*
*   If one of the #EXIF exits was taken, this code is never
*   processed.
           :
         #EDO
```

- In the following #DO TIMES example, a BCT loop is processed the number of times specified by the value in register 2 (R2). The loop count was previously loaded in R2. R2 is used for a BCT instruction and contains successive values down to 1. On exit from the #EDO, R2 is 0.

```
         #DO  TIMES=(R2)
           :
*   Code to process the number of times specified by the
*   value in R2.
           :
         #EDO
```

- In the following #DO TIMES example, a BCTR loop is processed 100 times. R2 and R3 are used for a BCTR instruction. R2 contains successive values 100, 99, 98, and so on down to 1; R3 holds the address of the start of the loop for the BCTR. On exit from the #EDO, R2 is 0.

```
         #DO  TIMES=(R2,100,R3)
           :
*   Code to process 100 times.
           :
         #EDO
```

- In the following #DO TIMES example, the value in EBW000 is loaded in R15 and a BCT loop is generated.

```
         #DO  TIMES=(R15,X/EBW000)
           :
*   Code to process the number of times specified by the value in
*   the byte at EBW000.
           :
         #EDO
```

- In the following #DO FROM example, a simple BC loop is generated. The processing is performed at least once with R3=0, then successively (with 4, 8, 12, and so on) up to and including the last value less than or equal to the contents of fullword in EBW000. For example,
  - If EBW000 contains 16, the last iteration is done with R3=16.
  - If EBW000 contains 15, the last iteration is done with R3=12.

```
         #DO  FROM=(R3,0),BY=4,TO=(EBW000)
           :
*   Code to process the specified number of times.
           :
         #EDO
```

- In the following #DO FROM example, a BC loop is generated that excludes the ending loop count. The processing is performed at least once with R3 equal to the contents of the fullword EBW000. At the end of each iteration, R3 is incremented by the length of field FL and compared to halfword EBW020. The loop continues as long as R3 is less than EBW020. For example, if EBW000 contains 10, EBW020 contains 100, and L'FL is 10, the iterations are performed with R3=10, 20, 30, and so on up to 90, but not 100.

```
         #DO  FROM=(R3,EBW000),BY=L'FL,TO=(H/EBW020,EXCLUSIVE)
           :
*   Code to process the specified number of times.
           :
         #EDO
```

- In the following #DO FROM example, a BXLE loop is generated. Processing is performed 10 times with R2 successively equal to 10, 20, 30, and so on up to 100.

```
         #DO  FROM=(R2,10),BY=(R4,10),TO=(R5,100)
           :
*   Code to process 10 times.
           :
         #EDO
```

- In the following #DO FROM example, a BXH loop is generated. Processing is performed at least once, with R2 successively equal to 100, 96, 92, and so on down to the last value that is greater than the contents of R5. R5 is initialized with an `LA  R5,VAL` instruction. For example, if R5 contains 3, the processing is done 25 times with 4 as the last value used in R2.

```
         #DO  FROM=(R2,100),BY=(R4,-4),TO=(R5,A/VAL)
           :
*   Code to process the specified number of times.
           :
         #EDO
```

- In the following #DO FROM example, a BXLE loop is generated and registers are saved during processing. Processing is performed at least once, with R3 successively equal to 1000, 1020, 1040, and so on up to the last value less than or equal to the contents of R15 (initialized from fullword EBW048). R5 is initialized with an `LA R5,VAL` instruction. The registers are saved during loop processing and reloaded by #ELOP for the BXLE:
  - R3 in EBW040
  - R14 in EBW044
  - R15 in EBW048.

  If there is no #ELOP macro, the #EDO macro reloads the registers.

  ```
           #DO   FROM=(R3,1000,EBW040),BY=(R14,20,EBW044),TO=(R15,EBW048)
             :
  *   Code to process the specified number of times.
             :
           #EDO
  ```

- In the following #DO FROM example, a BXLE loop is generated. In this example, the BXLE parameter is required because the type of loop cannot be determined from the parameters.

  ```
           #DO  BXLE,FROM=(R2),BY=(R4),TO=(R5)
             :
  *   Code to process using a BXLE loop based on the contents of
  *   R2, R4, and R5.
             :
           #EDO
  ```

- In the following example, the WHILE and UNTIL parameters are used together.

  ```
           #DO   WHILE=(LTR,R0,R0,Z),UNTIL=(CR,R2,LE,R3)
  *
  *   The WHILE condition is tested here.
             :
  *   loop processing
             :
  *   The UNTIL condition is tested here.
  *
           #EDO
  ```

- In the following example, the WHILE and TIMES parameters are used together.

  ```
           #DO   WHILE=(CR,R2,EQ,R3),TIMES=(R14,10)
  *
  *   The WHILE condition is tested here.
             :
  *   loop processing
             :
  *   The TIMES BCT is generated here.
  *
           #EDO
  ```

- In the following example, the WHILE, UNTIL, and TIMES parameters are used together.

  ```
           #DO   WHILE=(R3,NZ),UNTIL=(R2,EQ,R4),TIMES=(R5,#ITEMS)
  *
  *   The WHILE condition is tested here.
             :
  *   loop processing
             :
  *   The UNTIL condition is tested here.
  ```

```
*
*   The TIMES BCT is generated here.
*
         #EDO
```

Related Information

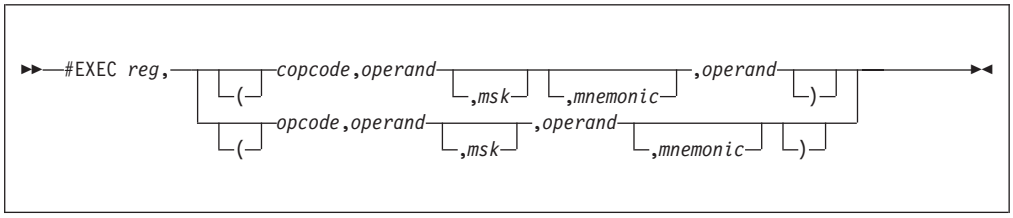
"#EXEC–Execute Macro" on page 58.

#EXEC–Execute Macro

Use this macro to generate an EX instruction. See *ESA/370 Principles of Operation* or *ESA/390 Principles of Operation* for more information about the EX instruction.

Note: You can also use the #EXEC macro as a conditional expression; see "Conditional Expression Format" on page 14 for more information about the syntax of a conditional expression.

Format



reg
is the register to use as the first operand in the EX instruction.

copcode
is an operation code starting with C; for example, CLC, CR, and so on.

opcode
is an operation code that sets a condition code but does not start with C; for example, TM, OC, and so on.

operand
is an operand for the instruction, which can be anything that the assembler language allows for an assembler instruction operand. See *ESA/370 Principles of Operation* or *ESA/390 Principles of Operation* for more information about assembler instructions and valid operands. Some of the types of operands include:

- A register, in the form R*n*, where *n* is the register number; for example, R1.
- A label refers to a defined area of storage (DS), a defined constant (DC), or an equated value. Generally, the size and type of label is implied from the type of instruction. However, you can force the size and type by prefixing the label as follows:

  A*label*
  specifies a 4-byte address contained at location *label*.

  F*label*
  specifies a fullword starting at location *label*.

  H*label*
  specifies a halfword starting at location *label*.

  X*label*
  specifies a byte starting at location *label*.

  I*label*
  specifies a 1-byte equated value.

  P*label*
  specifies packed data at location *label*.
- A numeric value, which must be an integer.
- A literal, for example:

  - A fullword (=F"1000")
  - A halfword (=H"10")
  - Characters (=C"HELLO").

  See *ESA/370 Principles of Operation* or *ESA/390 Principles of Operation* for more information about literals.
- An equate.
- An immediate value, which is a string that represents 1 byte and can be one of the following:
  - Character (C"A")
  - Hexadecimal (X'40')
  - Binary (B'10101010')
  - Length (L'EBW000).
- An arithmetic expression, which is an expression that resolves into an arithmetic value. For example:
  - 10 + 3
  - FLD + 10

msk
is the M3 or R3 operand of an RS instruction. See *ESA/370 Principles of Operation* or *ESA/390 Principles of Operation* for more information about the RS instruction.

Note: You ***must*** specify the *msk* with an RS instruction. Do ***not*** specify the *msk* for a non-RS instruction.

mnemonic
is one of the condition-code mnemonics shown in Table 2 on page 15.

Note: You ***must*** specify the *mnemonic* when you use the #EXEC macro as a conditional expression. Do ***not*** specify the *mnemonic* when using the #EXEC macro alone.

Entry Requirements

None.

Return Conditions

Control is returned to the next sequential instruction.

Programming Considerations

- All labels used in the SPM conditional expression can be no more than 32 characters long. Any additional characters are truncated.
- The register specified for *reg* is used as the first operand in the EX instruction. The rest of the expression is the subject instruction of the EX instruction and the branching condition. See *ESA/370 Principles of Operation* or *ESA/390 Principles of Operation* for more information about the EX instruction.

Examples

- In the following example, the #EXEC macro is used as a conditional expression for a #IF statement.

```
         #IF   #EXEC,R2,TM,FLAGS,X'00',ON
          :
*   Code to process
          :
         #EIF
```

- In the following example, the #EXEC macro is used to process a Move Character (MVC) instruction.

```
         #EXEC R5,MVC,EBW008(0),EBW008
          :
```

- In the following example, the #EXEC macro is used to process a Compare Logical Character under Mask (CLM) instruction.

```
         #EXEC R5,CLM,R6,0,0(R14)
          :
```

Related Information

- “#DO Macro Group” on page 48
- “#GOTO Macro Group” on page 61
- “#IF Macro Group” on page 63.

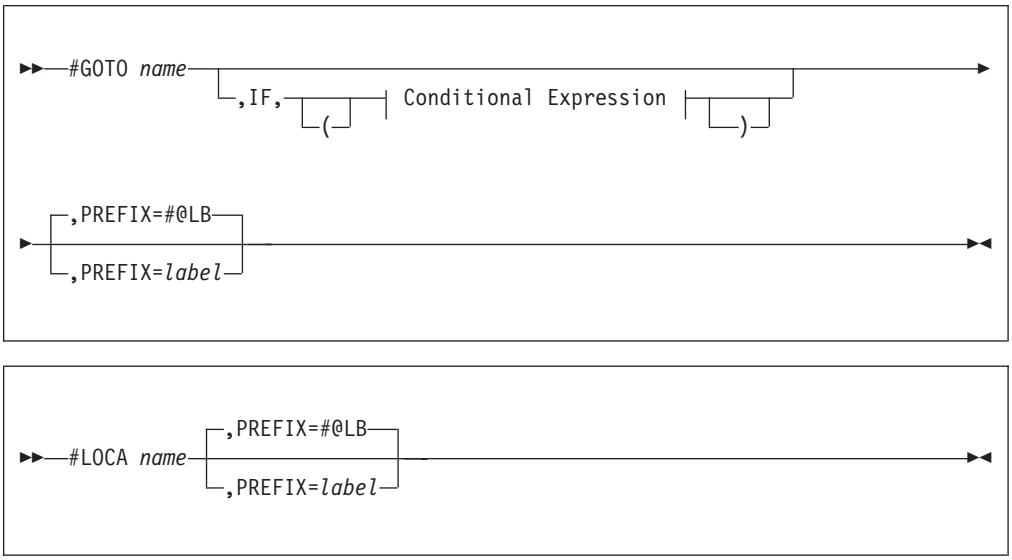
#GOTO Macro Group

Use this macro group to pass control to another section of code.

The #GOTO macro group includes the following macros:
- #GOTO
- #LOCA.

See "#GOTO Macro Group Processing" on page 85 for a diagram that shows the processing flow of the #GOTO macro group.

Format



#GOTO
generates a branch to the code starting with a corresponding #LOCA macro.

name
is a name assigned to the section of code associated with the #LOCA macro.

IF specifies when to take the branch condition based on a conditional expression. See "Conditional Expression Format" on page 14 for information about the syntax of a conditional expression.

PREFIX=*label*
specifies a prefix for all link labels generated by this macro group, where *label* is a 4-character alphabetic name.

#LOCA
specifies the start of the code for processing.

Entry Requirements

None.

Return Conditions

- For the #GOTO macro, control is passed to the next instruction if the conditional expression is false. Otherwise, control is passed to the corresponding #LOCA macro.
- For the #LOCA macro, control is passed to the next instruction.

Programming Considerations

- All labels used in the SPM conditional expression can be no more than 32 characters long. Any additional characters are truncated.
- Each macro statement and assembler instruction must begin on a new line in the application.
- The SPMs generate assembler instructions according to the specified macro parameters. The necessary branch instructions and link labels are generated internally to support the nonsequential processing. The SPMs generate standard link labels in the following format:

  ```
  #@LBn  EQU  *
  ```

  Where:
  #@LB is the link-label prefix.
  n is a sequence number that is generated automatically.

  You can use the PREFIX parameter to change the standard link-label prefix.
- The labels generated by macros nested inside a structure have the standard prefix #@LB unless another PREFIX parameter is specified with the inner SPMs.
- The TPFDF macros include an ERROR parameter that causes a branch to a #LOCA macro if an error occurs. You ***must*** code a #LOCA macro if you specify the ERROR parameter on a TPFDF macro.

  See the *TPFDF Programming Concepts and Reference* for more information about the TPFDF macros.
- You must code a #LOCA macro if you code a #GOTO macro.
- You can have multiple #GOTO macro statements for 1 #LOCA macro statement.
- You can code the #GOTO and the #LOCA macros statements in any order.

Examples

In the following example, a conditional expression is used to control the exit processing.

```
         #GOTO   XSR1ERR1,IF,(LTR,R0,R0,Z)
          :
*    Code to process if R0 is not zero (the condition is false).
          :

         #LOCA   XSR1ERR1
          :
*    Code to process if R0 is zero (the condition is true).
          :
```

Related Information

"#EXEC–Execute Macro" on page 58.

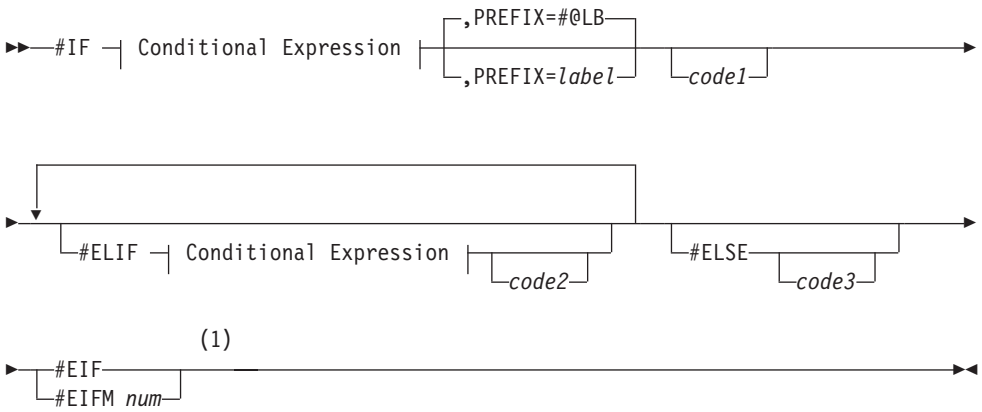
#IF Macro Group

Use this macro group to process specific code based on a set of conditions. The #IF macro group includes the following macros:

- #IF
- #ELIF
- #ELSE
- #EIF
- #EIFM.

See "#IF Macro Group Processing" on page 84 for a diagram that shows the processing flow of the #IF macro group.

Format



Notes:

1 Only code the #EIFM macro when you are nesting #IF structures. The #EIF macro is not required for every #IF macro if you use the #EIFM macro.

#IF

specifies the start of the #IF structure based on a conditional expression. Any code immediately following this macro is processed if the conditional expression is true. See "Conditional Expression Format" on page 14 for information about the syntax of a conditional expression.

PREFIX=*label*

specifies a prefix for all link labels generated by this macro group, where *label* is a 4-character alphabetic name.

code1

is the code to process when the #IF condition is true.

#ELIF

specifies the start of an additional selection (referred to as *else if*) based on a conditional expression. Any code immediately following this macro is processed when the #IF condition and any previous #ELIF conditions are false and the #ELIF condition is true.

See “Conditional Expression Format” on page 14 for information about the syntax of a conditional expression.

code2
is the code to process when the #ELIF condition is true.

#ELSE
specifies the start of the *else* selection. Any code that follows the #ELSE macro is processed when none of the #IF or #ELIF conditions are true.

code3
is the code to process when none of the #IF or #ELIF conditions are true.

#EIF
ends the #IF structure.

#EIFM *num*
generates multiple #EIF macro statements, where *num* is the number #EIF macros to generate. Use this when multiple nested #IF structures all end at the same location.

Entry Requirements

None.

Return Conditions

Control is returned to the next sequential instruction after the #EIF macro statement unless another assembler instruction or macro passes control outside of the #IF structure.

Programming Considerations

- All labels used in the SPM conditional expression can be no more than 32 characters long. Any additional characters are truncated.
- The #IF, #ELIF, #ELSE, #EIF, and #EIFM macros can only be used with the #IF macro group.
- Each macro statement and assembler instruction must begin on a new line in the application.
- A section of code (represented by *code1*, and so on) can consist of any number of standard assembler instructions, including other SPMs or assembler macros.
- The SPMs generate assembler instructions according to the specified macro parameters. The necessary branch instructions and link labels are generated internally to support the nonsequential processing. The SPMs generate standard link labels in the following format:

  ```
  #@LBn  EQU  *
  ```

  Where:
  #@LB is the link-label prefix.
  n is a sequence number that is generated automatically.

  You can use the PREFIX parameter to change the standard link-label prefix.
- The labels generated by macros nested inside a structure have the standard prefix #@LB unless another PREFIX parameter is specified with the inner SPMs.
- Only one section of code is performed each time the #IF structure is processed.
- You can nest several #IF structures. When nesting, each nested #IF structure must be completely contained in the previous #IF structure. When processed, the #ELSE macro will be tied to the most recent #IF macro.

Examples

- In the following example, an explicit #ELIF macro and #ELSE macro are used to specify alternate selections.

```
         #IF   (CLI,XSREST,EQ,1)
          :
*   Code to process if the field XSREST = 1
          :
         #ELIF (TM,XSFLAG,X'80',ON)
          :
*   Code to process if the field XSREST does not equal 1,
*   but bit 0 of XSFLAG is ON.
          :
         #ELSE
          :
*   Code to process for all other cases,
*   XSREST does not equal 1 and bit 0 of XSFLAG is OFF.
          :
         #EIF
          :
*   Processing continues here after one (and only one) of
*   the above processes has been performed.
          :
```

- In the following example, the #ELSE macro is not specified because there is no specific processing required if R2 < R3.

```
         #IF   CR,R2,GE,R3
          :
*   Code to process if R2 >= R3.
          :
         #EIF
```

- In the following example, the #EIFM macro is used to end several nested #IF structures. Also, note that the #ELSE macro is associated with the third #IF macro statement.

```
         #IF   (CLI,XSREST,EQ,1)
           :
*   Code to process if the field XSREST = 1.
            :
             #IF   CR,R2,GE,R3
               :
*   Code to process if the field XSREST = 1 and R2 >= R3.
                :
                 #IF   CR,R4,LT,R5
                   :
*   Code to process if the field XSREST = 1, R2 >= R3, and R4 < R5.
                    :
                 #ELSE
                :
*   Code to process if the field XSREST = 1 and R2 >= R3,
*   but R4 >= R5.
            :
         #EIFM 3
           :
*   Processing continues here.
          :
```

Related Information

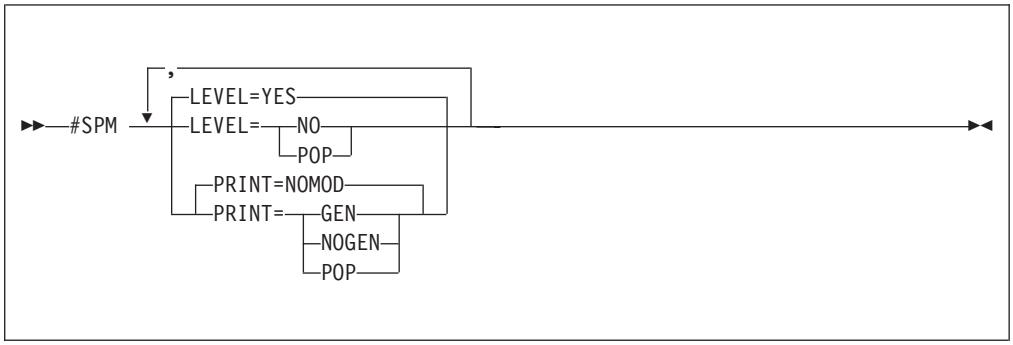
"#EXEC–Execute Macro" on page 58.

#SPM–Assembly Output Processing

Use this macro to control the printing of the following:

- Nesting-level indication
- Link-label indication (#@LB*nn*)
- Structured programming macro (SPM) expansions.

Format



LEVEL
controls the printing of assembler messages that indicate the nesting level and link labels, where:

YES
prints the assembler messages.

NO
does not print the assembler messages.

POP
restores the assembler message printing to the state that was set before the previous #SPM macro with the LEVEL parameter.

PRINT
controls the printing of the SPM macro expansions, where:

GEN
prints the macro expansions.

NOGEN
does not print the macro expansions.

NOMOD
cancels any previous #SPM PRINT=GEN or #SPM PRINT=NOGEN statement.

POP
restores the macro expansion printing to the state that was set before the previous #SPM macro with the PRINT parameter.

Entry Requirements

None.

Return Conditions

Control is returned to the next sequential instruction.

Programming Considerations

None.

Examples

- The following is an example of the PRINT parameter.

```
#SPM PRINT=GEN
```

  When PRINT=GEN is coded, each SPM (except #) generates the following code at the beginning of the macro.

```
PUSH  PRINT
PRINT GEN
```

  The following code is generated at the end of each macro:

```
POP   PRINT
```

- The following example shows how the nesting levels, link labels, and macro expansions are printed.

```
             #SPM  PRINT=GEN,LEVEL=YES
             #DO   WHILE=(CR,R2,EQ,R3),PREFIX=AAAA
+            PUSH  PRINT
+            PRINT GEN
+AAAA3       EQU   *
+            CR    R2,R3
+            BC    15-8,AAAA2
+*,1
+            POP   PRINT
                #DO   WHILE=(CR,R3,EQ,R4)
+            PUSH  PRINT
+            PRINT GEN
+#@LB9       EQU   *
+            CR    R3,R4
+            BC    15-8,#@LB8
+*,2
+            POP   PRINT
                   #IF   ICM,R2,7,EBW000,NZ
+            PUSH  PRINT
+            PRINT GEN
+            ICM   R2,7,EBW000
+            BC    15-7,#@LB14             BR FALSE TO #ELSE OR #EIF
+*,3
+            POP   PRINT
                      LA  R4,0
                   #EIF
+            PUSH  PRINT
+            PRINT GEN
+#@LB14     EQU   *
+*,2
+            POP   PRINT
                #EDO
+            PUSH  PRINT
+            PRINT GEN
+            BC    15,#@LB9
+#@LB8        EQU   *
+            POP   PRINT
+            PUSH  PRINT
+            PRINT GEN
+*,1
+            POP   PRINT
               LA  R0,0
             #EDO
+            PUSH  PRINT
+            PRINT GEN
```

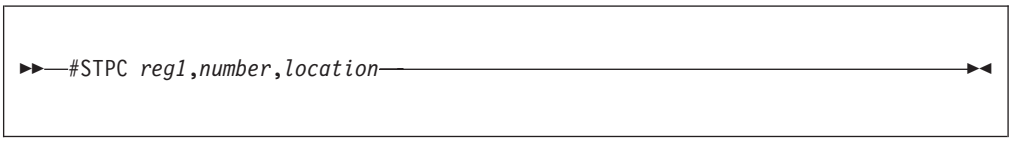
Related Information

None.

#STPC–Step a Byte or Character

Use this macro to generate inline code to increment or decrement (also referred to as *step*) a 1-byte value at a specified location.

Format



reg1
is a working register for the macro.

number
is the amount by which to increment or decrement the specified value. If you specify a positive number, the value at the specified location will be incremented. If you specify a negative number, the value at the specified location will be decremented.

location
is the label of the area that contains the value to be incremented or decremented.

Entry Requirements

None.

Return Conditions

- The contents of *reg1* are overwritten.
- Control is returned to the next sequential instruction.

Programming Considerations

- All labels used in the SPM conditional expression can be no more than 32 characters long. Any additional characters are truncated.
- In a tightly coupled environment, use care when changing common storage. It is possible that more than one program can change a value at the same time. Use the #STPF macro when updating shared storage to ensure the field is updated consistently.

Examples

- In the following example, the value in the byte at EBW033 is incremented by 10.

```
#STPC R0,10,EBW033
```

- In the following example, the value in the byte at EBX008 is decremented by 4.

```
#STPC R14,-4,EBX008
```

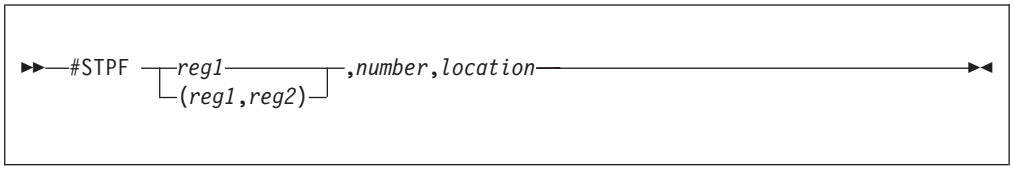
Related Information

- “#STPF–Step a Fullword” on page 70
- “#STPH–Step a Halfword” on page 72
- “#STPR–Step Registers” on page 73.

#STPF–Step a Fullword

Use this macro to generate inline code to increment or decrement a fullword value at a specified location. You can also use this macro to step shared common storage.

Format



reg1
is a working register for the macro.

reg1,reg2
are a pair of registers used by the Compare and Swap (CS) instruction. See *ESA/370 Principles of Operation* or *ESA/390 Principles of Operation* for more information about the CS instruction. Use this form to change the contents of a fullword in shared storage to ensure the field is always updated consistently.

number
is the amount by which to increment or decrement the specified value. If you specify a positive number, the value at the specified location will be incremented. If you specify a negative number, the value at the specified location will be decremented.

location
is the label of the area that contains the value to be incremented or decremented.

Entry Requirements

None.

Return Conditions

- The contents of *reg1* are overwritten.
- Control is returned to the next sequential instruction.

Programming Considerations

- All labels used in the SPM conditional expression can be no more than 32 characters long. Any additional characters are truncated.
- In a tightly coupled environment, use care when changing common storage. It is possible that more than one program can change a value at the same time. Use a pair of registers (*reg1*,*reg2*) with this macro when changing common storage to avoid this problem.

Examples

- In the following example, the value in the fullword at EBX004 is incremented by 100000.

```
#STPF R2,100000,EBX004
```

- In the following example, the value in the fullword at EBX036 is decremented by 760.

```
#STPF R5,-760,EBX036
```

- In the following example, the value in the fullword at the shared location called SHARED is incremented by 20.

```
#STPF (R2,R3),20,SHARED
```

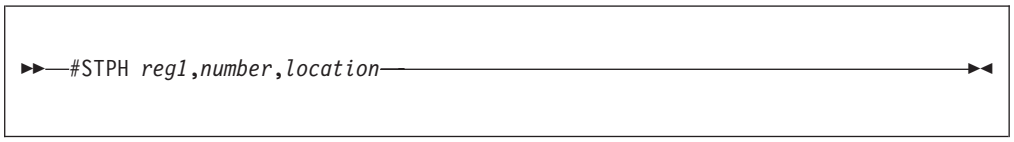
Related Information

- "#STPC–Step a Byte or Character" on page 69
- "#STPH–Step a Halfword" on page 72
- "#STPR–Step Registers" on page 73.

#STPH–Step a Halfword

Use this macro to generate inline code to increment or decrement a halfword value at a specified location.

Format



reg1
is a working register for the macro.

number
is the amount by which to increment or decrement the specified value. If you specify a positive number, the value at the specified location will be incremented. If you specify a negative number, the value at the specified location will be decremented.

location
is the label of the halfword area that contains the value to be incremented or decremented.

Entry Requirements

None.

Return Conditions

- The contents of *reg1* are overwritten.
- Control is returned to the next sequential instruction.

Programming Considerations

- All labels used in the SPM conditional expression can be no more than 32 characters long. Any additional characters are truncated.
- In a tightly coupled environment, use care when changing common storage. It is possible that more than one program can change a value at the same time. Use the #STPF macro when updating shared storage to ensure the field is updated consistently.

Examples

- In the following example, the value in the halfword at EBW002 is incremented by 300.

```
#STPH R4,300,EBW002
```

- In the following example, the value in the halfword at EBW086 is decremented by 400.

```
#STPH R7,-400,EBW086
```

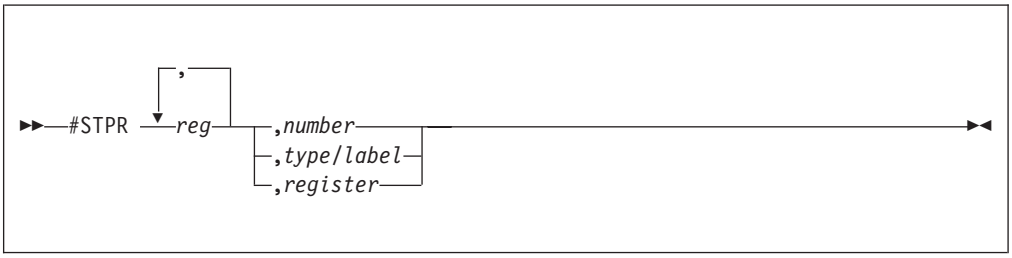
Related Information

- “#STPC–Step a Byte or Character” on page 69
- “#STPF–Step a Fullword” on page 70
- “#STPR–Step Registers” on page 73.

#STPR–Step Registers

Use this macro to generate inline code to increment or decrement 1 or more registers.

Format



reg
is a register that contains the value to be incremented or decremented. You can specify as many as 10 registers.

number
is the amount by which to increment or decrement the specified value. If you specify a positive number, the value in the specified register will be incremented. If you specify a negative number, the value in the specified register will be decremented.

type|*label*
specifies the amount by which to increment or decrement the specified value, where *label* is the label of a location that contains the amount and is prefixed by one of the following:

A|*label*
specifies a 4-byte address contained at location *label*.

F|*label*
specifies a fullword starting at location *label*.

H|*label*
specifies a halfword starting at location *label*.

X|*label*
specifies a byte starting at location *label*.

I|*label*
specifies a 1-byte equated value.

P|*label*
specifies packed data at location *label*.

register
is a register that contains the amount by which to increment or decrement the specified value.

Entry Requirements

None.

Return Conditions

- The contents of *reg* are overwritten.
- Control is returned to the next sequential instruction.

Programming Considerations

All labels used in the SPM conditional expression can be no more than 32 characters long. Any additional characters are truncated.

Examples

- In the following example, registers R0 and R4 are incremented by the contents of R7.

```
#STPR R0,R4,R7
```

- In the following example, registers R0 through R5 are incremented by the value of the fullword at location EBX000.

```
#STPR R0,R1,R2,R3,R4,R5,F/EBX000
```

- In the following example, the specified registers are incremented by the value of the halfword at location EBX032.

```
#STPR R0,R4,R14,R2,R5,R3,H/EBX032
```

- In the following example, registers R7 and R15 are incremented by 12.

```
#STPR R7,R15,12
```

- In the following example, registers R0 and R4 are decremented by 4.

```
#STPR R0,R4,-4
```

- In the following example, registers R14, R15 and R0 are incremented by the length of the field `GR00ALC`.

```
#STPR R14,R15,R0,L'GR00ALC
```

Related Information

- "#STPC–Step a Byte or Character" on page 69
- "#STPH–Step a Halfword" on page 72
- "#STPF–Step a Fullword" on page 70.

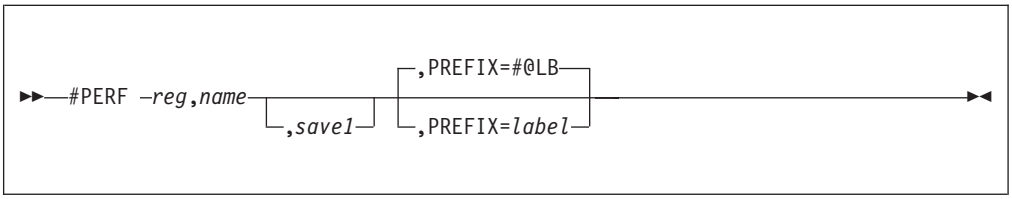
#SUBR Macro Group

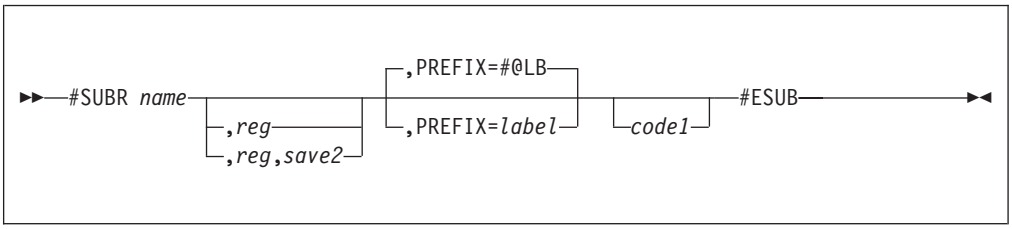
Use this macro group to call common subroutines. Generally, subroutines:
- Are logically self-contained entities
- Have one entry point
- Have one exit point
- Are not enclosed in any other logic structures.

The #SUBR macro group includes the following macros:
- #PERF
- #SUBR
- #ESUB.

See "#SUBR Macro Group Processing" on page 86 for a diagram that shows the processing flow of the #SUBR macro group.

Format





#PERF
branches to a specified subroutine.

#SUBR
specifies the start of the subroutine.

#ESUB
specifies the end of the subroutine. If *save2* was specified on the #SUBR macro, *reg* is reloaded before control is returned.

reg
is a work register used for branching to the subroutine. If the subroutine code is physically located before the corresponding #PERF macro, the *reg* parameter ***must*** be specified on the #SUBR macro as well.

The register specified on the #SUBR macro ***must*** be the same register specified on the #PERF macro. If there is a conflict between the registers, an assembly error is issued and the register specified on #SUBR is ignored.

name
is the name of the subroutine.

save1
is a fullword save area into which *reg* is stored before branching to the subroutine. On return from the subroutine, *reg* is reloaded from *save1*.

save2
is a fullword save area into which *reg* is stored for the duration of the subroutine processing. This allows the subroutine to use *reg* during processing.

If you do not specify *save2* on the #SUBR macro, *reg* is not saved.

PREFIX=*label*
specifies a prefix for all link labels generated by this macro group, where *label* is a 4-character alphabetic name.

code1
is the code to process for the subroutine.

Entry Requirements

None.

Return Conditions

Control is returned to the next sequential instruction.

Programming Considerations

- All labels used in the SPM conditional expression can be no more than 32 characters long. Any additional characters are truncated.
- The #PERF, #SUBR, and #ESUB macros can only be used with the #SUBR macro group.
- Each macro statement and assembler instruction must begin on a new line in the application.
- A section of code (represented by *code1*, and so on) can consist of any number of standard assembler instructions, including other SPMs or assembler macros.
- The SPMs generate assembler instructions according to the specified macro parameters. The necessary branch instructions and link labels are generated internally to support the nonsequential processing. The SPMs generate standard link labels in the following format:

  ```
  #@LBn  EQU  *
  ```

  Where:
  #@LB is the link-label prefix.
  n is a sequence number that is generated automatically.

  You can use the PREFIX parameter to change the standard link-label prefix.
- The labels generated by macros nested inside a structure have the standard prefix #@LB unless another PREFIX parameter is specified with the inner SPMs.
- You can have multiple #PERF macro statements for one #SUBR macro statement.
- Subroutines cannot be nested; that is, a subroutine cannot call another subroutine. A nested subroutine generates an assembly warning; however, the assembly continues normally.

Examples

In the following example, there is a call to subroutine XSR1RLCH.

```
        #PERF R14,XSR1RLCH,EBW080     Release old chain
*
*   R14 is stored in EBW080, the subroutine is executed,
*   then R14 is reloaded from EBW080.
*   Processing continues with R14 as it was before the
```

```
*   subroutine was invoked by #PERF
*   (assuming the subroutine itself did not corrupt
*   EBW080 through EBW083).
          :
          :
          :
*
         #SUBR XSR1RLCH,R14,EBW084   Chain-release subroutine
*
*   The return register (R14) is stored in EBW084 and can be
*   used in subroutine logic (but do not change the contents
*   of EBW080 through EBW087...).
          :
*   subroutine processing
          :
         #ESUB                        End of subroutine named XSR1RLCH
*                                     R14 is restored from EBW084
*                                     before the BR to return
```

Related Information

None.

TPFDF Structured Programming Macro Group Processing Diagrams

The following contains diagrams that show the flow of each of the following:

- Selection and iteration macro groups (#CASE, #DO, and #IF)
- Branch and subroutine macro groups (#GOTO and #SUBR).

Selection and Iteration Macro Groups

The following section shows the processing flow of the selection and iteration macro groups.

#CASE Macro Group Processing

Figure 5 on page 80 shows the processing flow of the #CASE macro group.

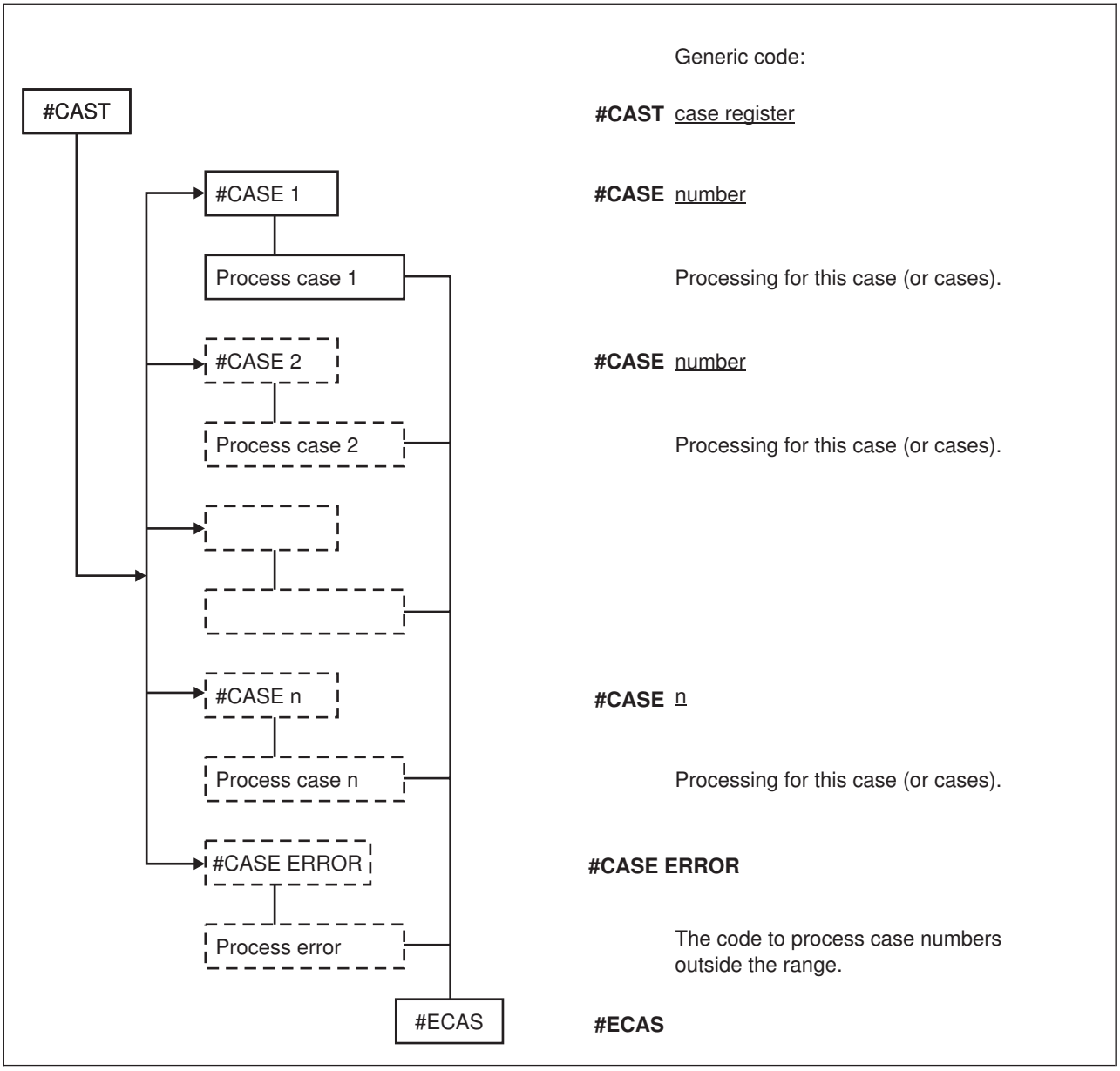


Figure 5. Selection: #CASE Macro Group. In this diagram, the boxes with broken lines are optional.

#DO Macro Group Processing

Figure 6 on page 81 shows the processing flow of a #DO WHILE loop.

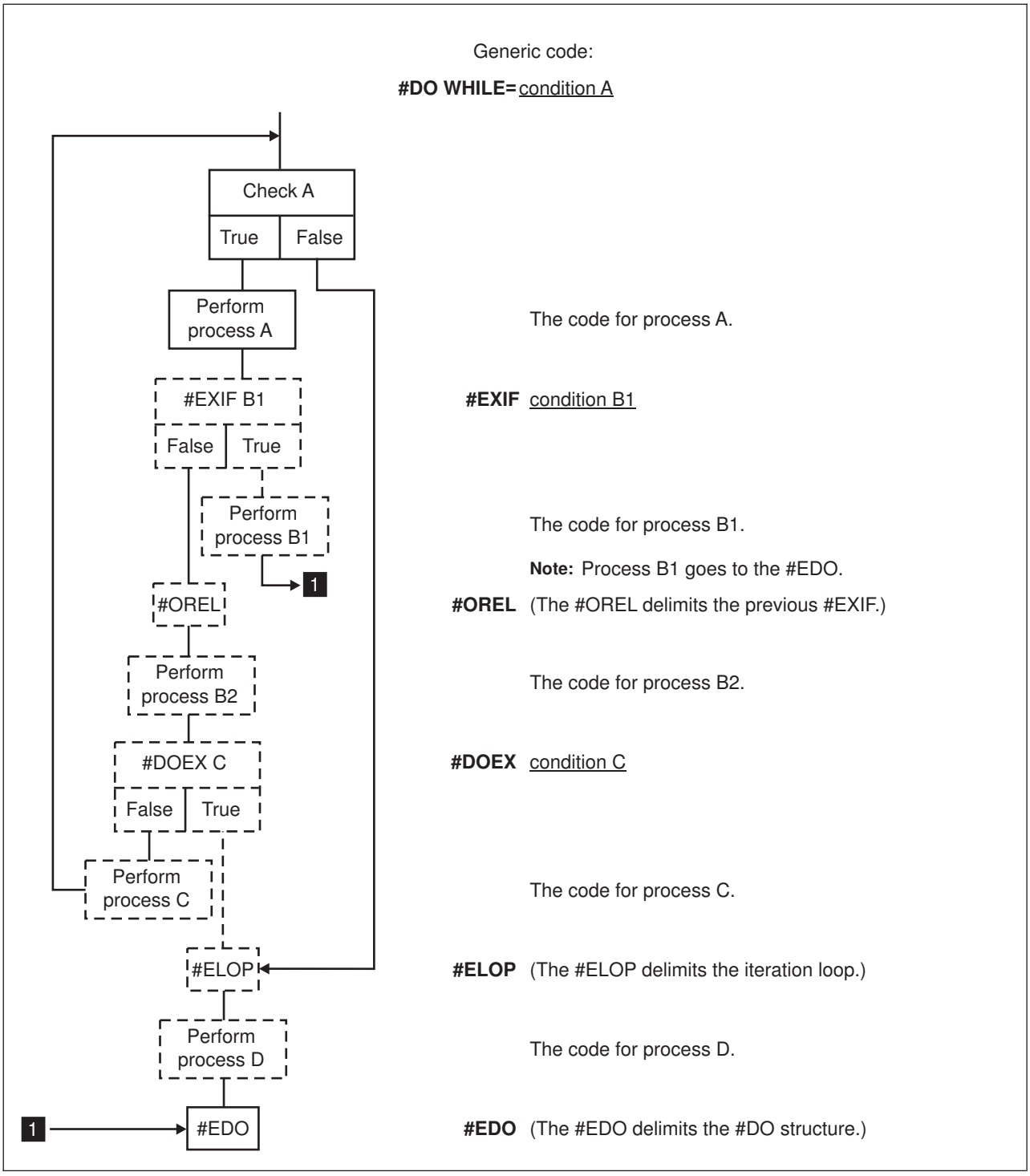


Figure 6. Iteration: #DO Macro Group with the WHILE Parameter. In this diagram, the boxes with broken lines are optional.

Figure 7 on page 82 shows the processing flow of a #DO UNTIL loop.

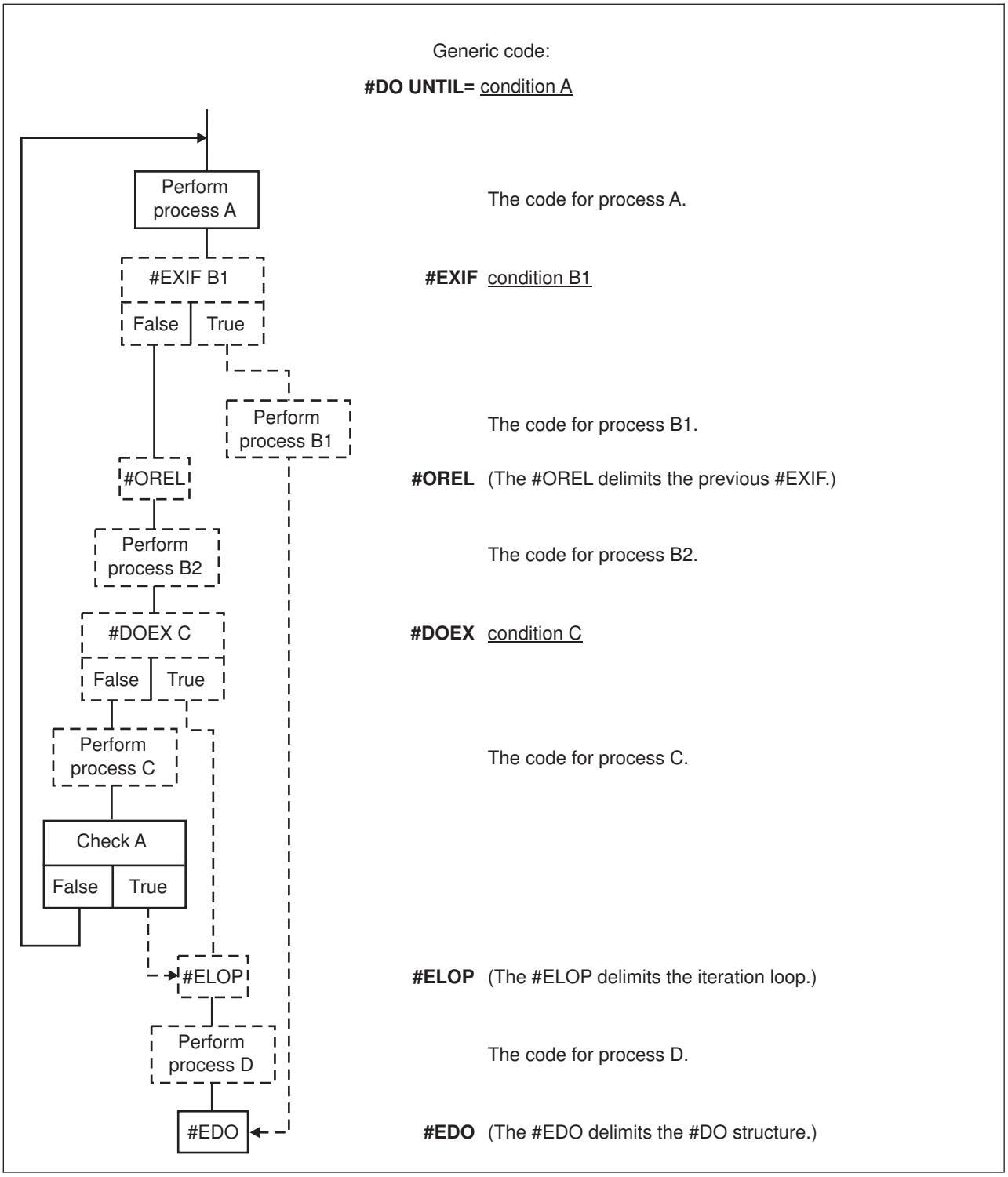


Figure 7. Iteration: #DO Macro Group with the UNTIL Parameter. In this diagram, the boxes with broken lines are optional.

Figure 8 on page 83 shows the processing flow of a #DO TIMES or #DO FROM loop.

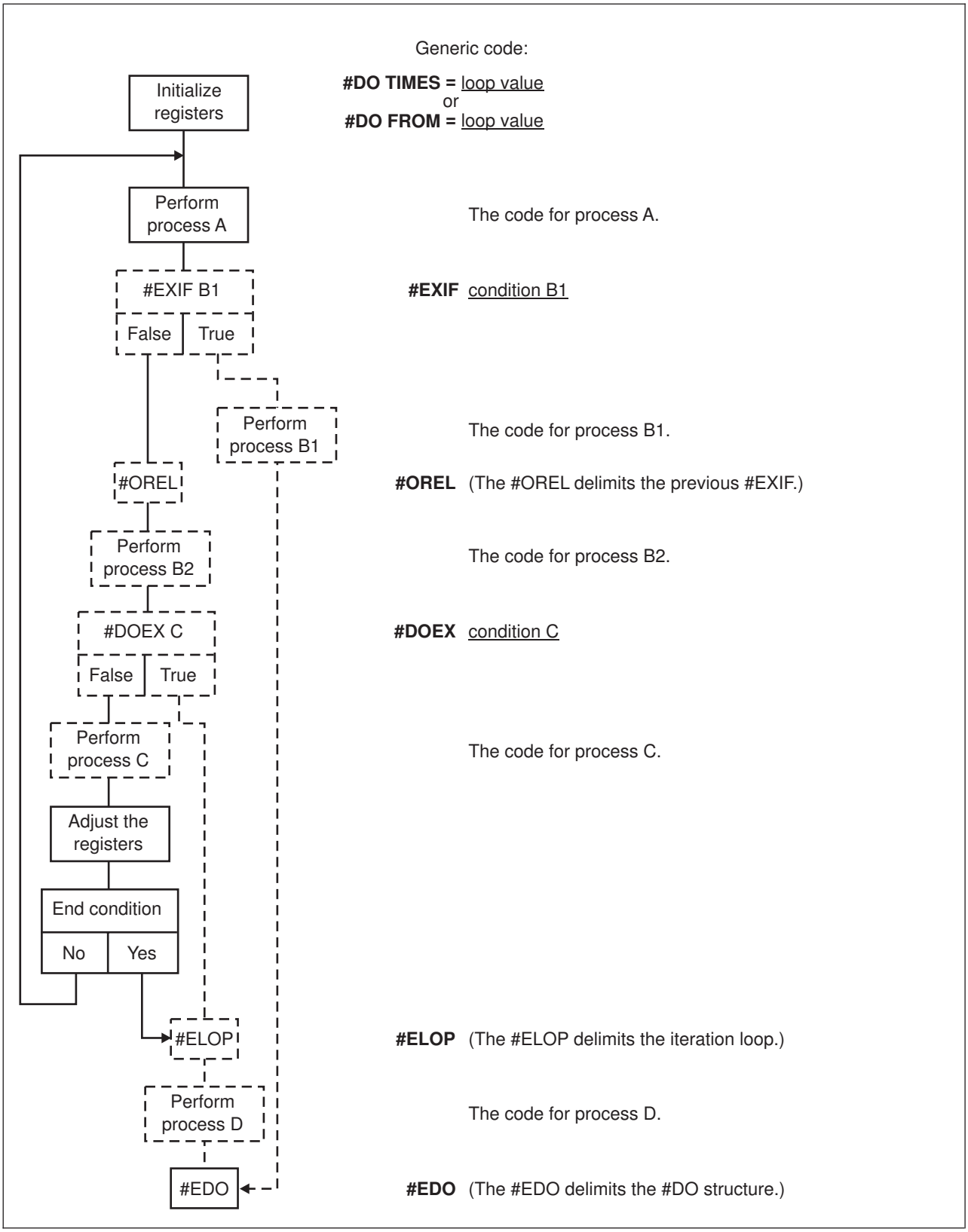


Figure 8. Iteration: #DO Macro Group with the FROM or TIMES Parameter. In this diagram, the boxes with broken lines are optional.

Figure 9 on page 84 shows the processing flow of a #DO INF loop.

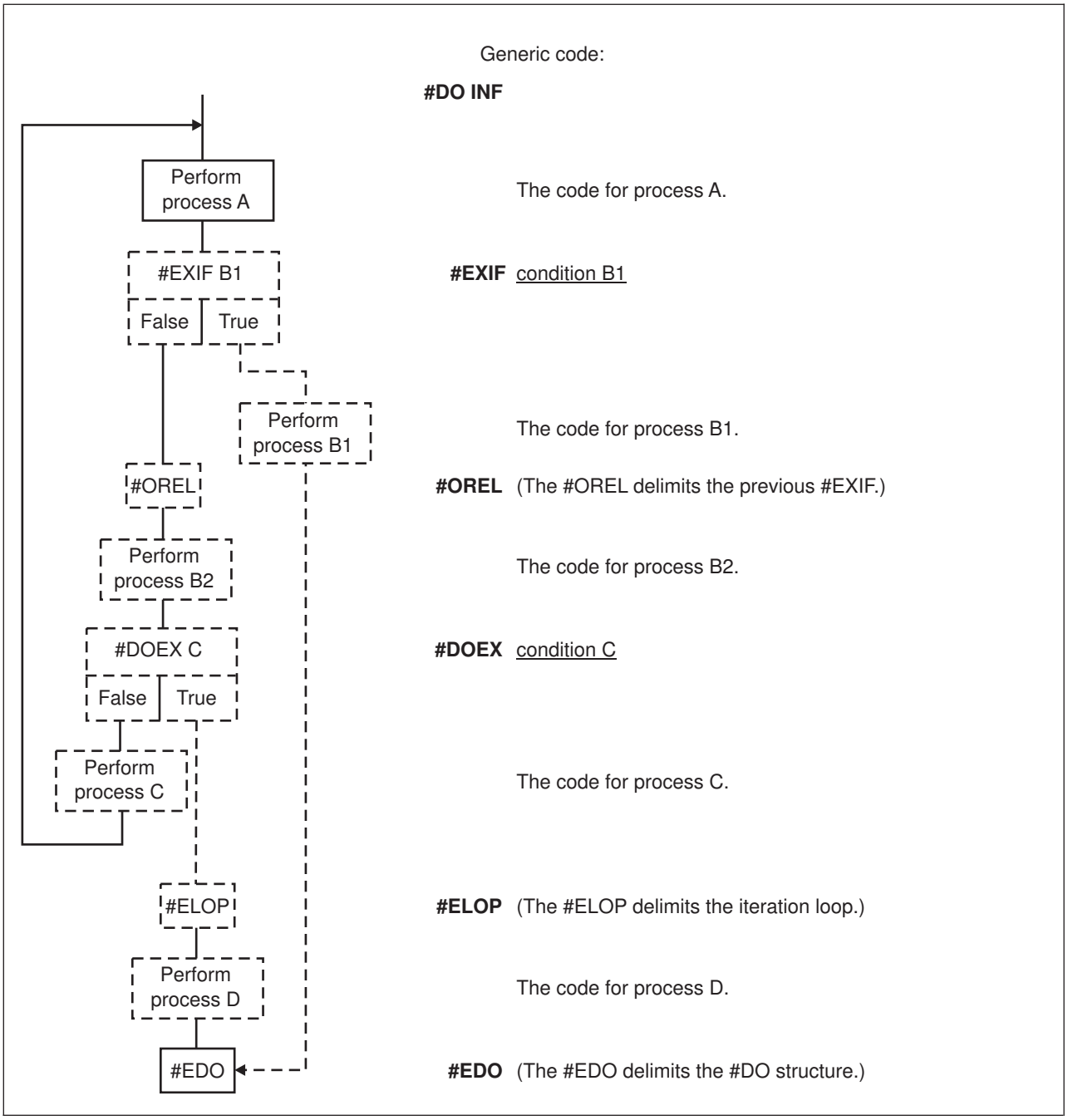


Figure 9. Iteration: #DO Macro Group with the INF Parameter. In this diagram, the boxes with broken lines are optional.

#IF Macro Group Processing

Figure 10 on page 85 shows the processing flow of the #IF macro group.

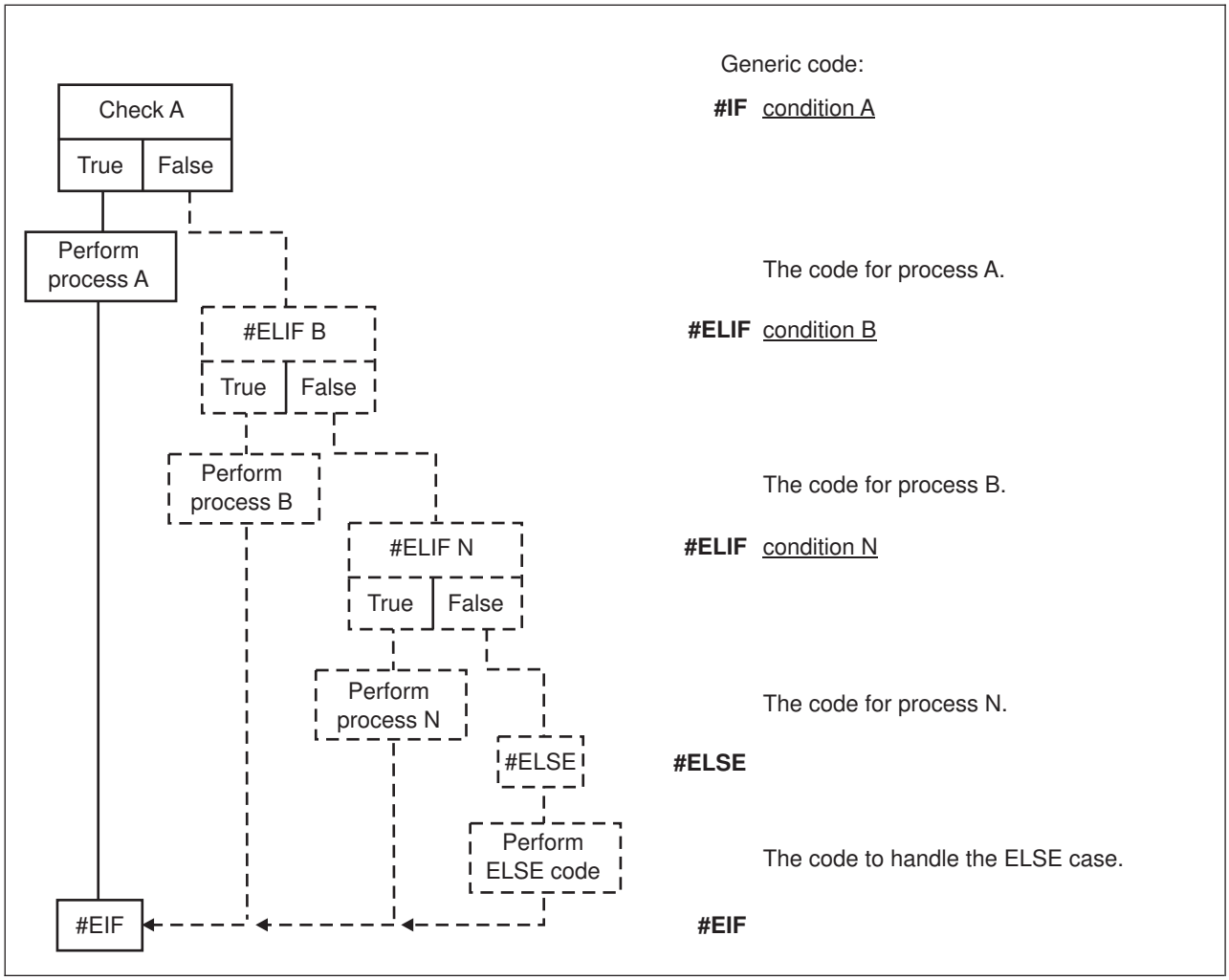


Figure 10. Selection: #IF Macro Group. In this diagram, the boxes with broken lines are optional.

Branch and Subroutine Macro Groups

The following section shows the processing flow of the branch and subroutine macro groups.

#GOTO Macro Group Processing

Figure 11 on page 86 shows the processing flow of the #GOTO macro group.

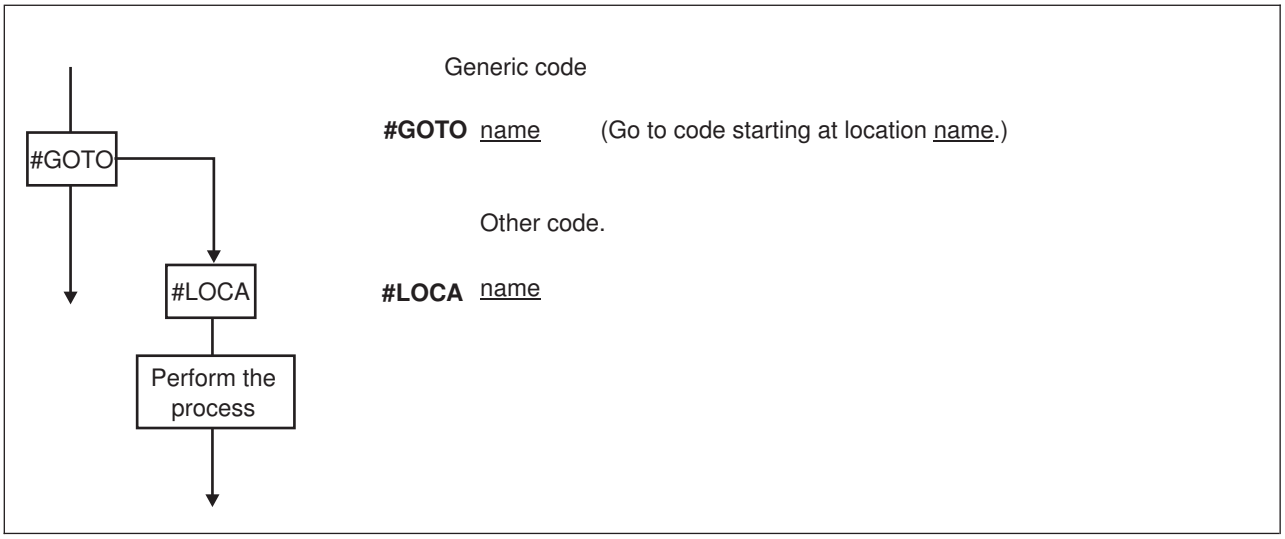


Figure 11. Branch: #GOTO Macro Group

#SUBR Macro Group Processing

Figure 12 shows the processing flow of the #SUBR macro group.

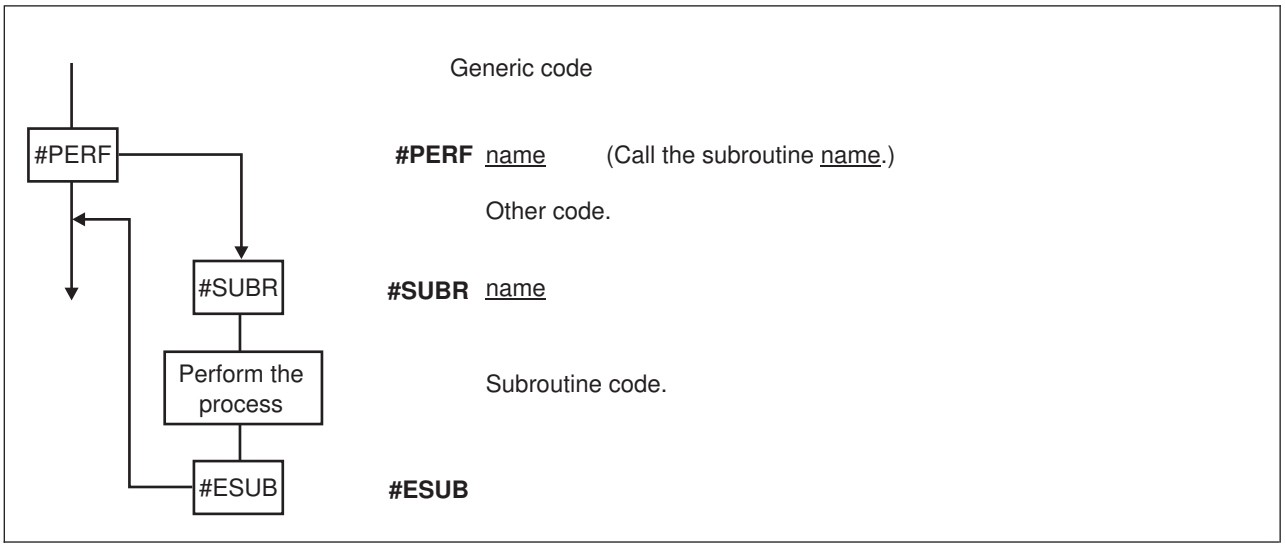


Figure 12. Subroutine: #SUBR Macro Group

Part 3. TPF Structured Programming Macros

TPF Structured Programming Macros: Reference

The following contains an alphabetic listing of the TPF structured programming macros (SPMs).

Note: These SPMs are available only with the TPF system. If you use TPFDF in an Airline Control System (ALCS) environment, see Part 2, “TPFDF Structured Programming Macros” on page 7 for information about the SPMs available for your environment.

The description of each SPM includes the following information:

Format
Provides a syntax (railroad track) diagram for the macro and a description of each parameter and variable. See “How to Read the Syntax Diagrams” on page x for more information about syntax diagrams.

Entry Requirements
Lists any special conditions that must be true when you use the macro.

Return Conditions
Lists what is returned when the macro is finished processing.

Programming Considerations
Lists any additional considerations for using the macro, including any restrictions or limitations.

Examples
Provides one or more examples that show you how to code the macro.

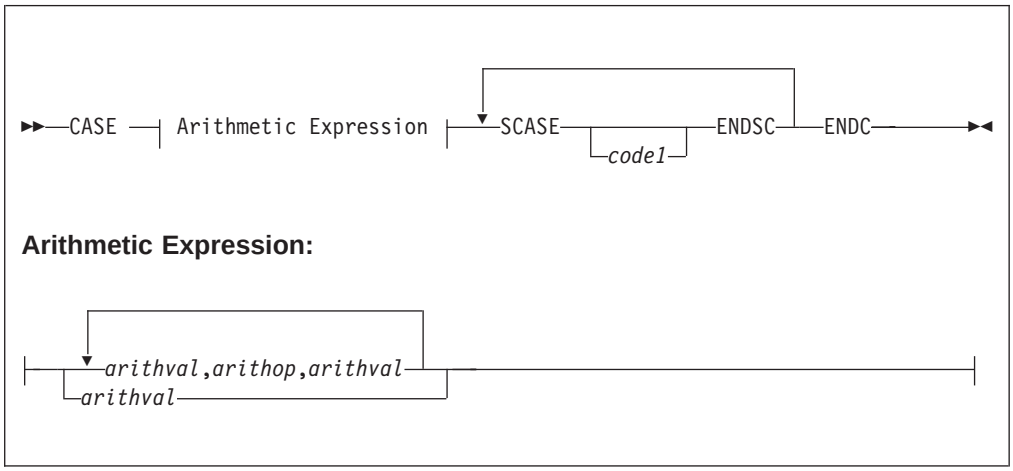
Related Macros
Lists where to find information about related macros.

CASE Macro Group

Use this macro group to select between several alternatives without using complex If-Then-Else logic. This macro group is used when the choice between a number of different code paths can be controlled by an arithmetic variable.

The CASE macro group includes the following macros:
- CASE
- SCASE
- ENDSC
- ENDC.

Format



CASE
starts a group of subcases based on an arithmetic expression.

If the arithmetic expression resolves to 0, the first subcase is processed; if the arithmetic expression resolves to 1, the second subcase is processed, and so on. There is no default subcase.

arithval
is a number represented directly in numeric form or in symbolic form. This value can be one of the following:
- Variable
- Constant
- Result of an arithmetic expression.

arithop
is one of the following arithmetic operators:

Operator	Description
+	Addition
−	Subtraction
*	Multiplication
/	Integer division
//	Remainder.

SCASE
specifies the start of a subcase.

code1
is the code to process for the associated subcase.

ENDSC
specifies the end of a subcase.

ENDC
specifies the end of the group of subcases.

Entry Requirements

None.

Return Conditions

- Control is returned to the next sequential instruction after the ENDC macro statement unless another assembler instruction or macro passes control outside the CASE structure.
- Any combination (or none) of the four work registers (default R0–R1, R14–R15) can be used by the CASE macro. A message is generated for each work register used. The contents of each work register used are unknown. The work registers can be changed by coding the WORK0 and WORK2 parameters of the DCL macro. Each of these parameters must be the even register of an even-odd pair. The contents of all other registers are preserved across this macro call.

Programming Considerations

- The CASE, SCASE, ENDSC, and ENDC macros can only be used with the CASE macro group.
- Each macro statement and assembler instruction must begin on a new line in the application.
- A section of code (represented by *code1*, and so on) can consist of any number of standard assembler instructions, including other SPMs or assembler macros.
- Because the SPMs are assembler language macros, all symbols used with the macros must be previously defined to the assembler. In addition, for the TPF SPMs, you must declare the attributes of the symbols using the DCL macro.
- If you do not code all the possible subcases, the results cannot be predicted.

Examples

- In the following example, the subcase is selected based on the result of 1+1; therefore, the subcase with `LA R1,3` is always selected.

```
CASE 1,+,1
   SCASE
      LA R1,1
   ENDSC
   SCASE
      LA R1,2
   ENDSC
   SCASE
      LA R1,3
   ENDSC
ENDC
```

- The following example shows the use of variables in the arithmetic expression.

```
NUM1     EQU   EBW012,4                    Reentrant for NUM1 DS A
NUM2     EQU   EBW024,4                    Reentrant for NUM2 DS A
         DCL NUM1,UNSIGNED
         DCL NUM2,UNSIGNED
           :
           :
*   Code that assigns values to NUM1 and NUM2
           :
```

```
            :
*
         CASE NUM1,-,NUM2
            SCASE
*                                    WHEN NUM1-NUM2 = 0
               :
            ENDSC
            SCASE
*                                    WHEN NUM1-NUM2 = 1
               :
            ENDSC
         ENDC
```

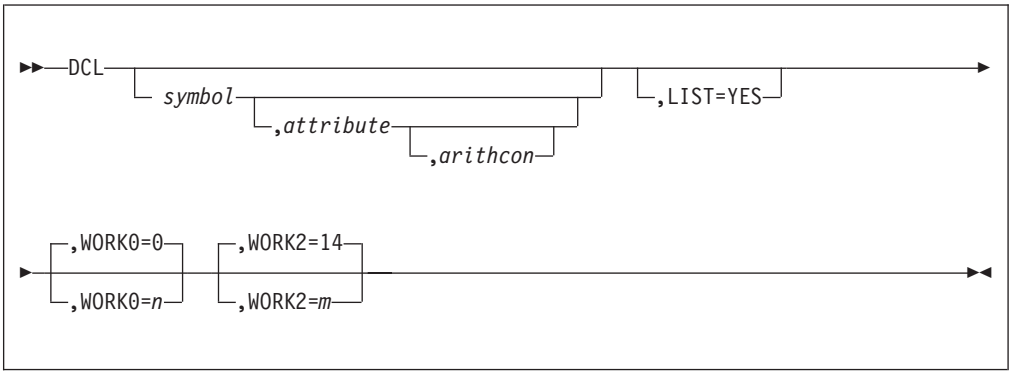
Related Information

- “DCL–Declare” on page 93
- “SELECT Macro Group” on page 117.

DCL–Declare

Use this macro to specify the attributes for symbols referred to in the IF, DO, CASE, and LET macros. These attributes remain in effect until they are changed by a subsequent DCL macro.

Format



symbol
is the label of a variable or constant that will be referred to in the structured programming macros (SPMs).

attribute
is one of the following types for the symbol being declared:

SIGNED
The attribute normally given to fullword (F) or halfword (H) symbols. The SIGNED attribute causes the field to be treated as containing a positive or negative fixed binary number whose high-order bit indicates the sign.

UNSIGNED
The attribute normally given to any symbol whose type is not a fullword (F), halfword (H), or character (C). The UNSIGNED attribute causes the field to be treated as containing a positive fixed binary number.

REGISTER
The attribute normally given to symbols defined to the assembler as follows:

```
symbol  EQU  n
```

The REGISTER attribute causes the symbol to be treated as a general-purpose register using *n* as the register number.

CHARACTER
The attribute normally given to character (C) symbols. The CHARACTER attribute causes the field to be treated as a string of bytes.

If you do not specify an attribute for a symbol, the symbol is removed from the symbol table.

arithcon
is a decimal integer that represents the length of the symbol, in bytes, unless REGISTER is specified as the attribute. In that case, it is the general-purpose register number. If you do not specify *arithcon*, the length (L') defined to the assembler is used. This length (L') is not available at assembly time under some assemblers if the symbol is generated by a macro. If REGISTER is

specified but the register number is not specified, the register defaults to the length of the symbol, or 1 if the symbol is an equate.

The value of the arithmetic constant is restricted by the attributes declared as follows:

Attribute	Value Range
SIGNED	1–4
UNSIGNED	1–4
CHARACTER	1–256
REGISTER	0–15

LIST=YES
prints all symbols that currently have their assembler attributes overridden with the DCL macro. The last attributes assigned are printed with the symbol.

WORK0=*n*
defines the first even-odd pair of general-purpose registers to be used by the SPMs. The value of *n* must be even. The registers are available for use in between macros but will be changed during macro processing.

WORK2=*m*
defines the second even-odd pair of general-purpose registers to be used by the structured programming macros. The value of *m* must be even and cannot be 0. The registers are available for use in between macros but may be changed during macro processing.

Entry Requirements

None.

Return Conditions

Control is returned to the next sequential instruction.

Programming Considerations

- Because the SPMs are assembler language macros, all symbols used with the macros must be previously defined to the assembler. In addition, for the TPF SPMs, you must declare the attributes of the symbols using the DCL macro.

  For example, to use a symbol in a conditional expression following the IF macro, you must define the symbol to the assembler and specify the symbol using the DCL macro.

  There are two ways to define a symbol to the assembler:
  – Use an EQUATE value.
  – Use DS to specify space if a reentrant program is not required.

  For example:

  ```
  NUM   EQU   EBX016,4,C'F'
  ```

  is a reentrant form of:

  ```
  NUM   DS    F
  ```

  To use NUM in the SPMs, the attributes for the NUM symbol must be declared using the DCL macro:

  ```
        DCL  NUM,SIGNED,4
  ```

The previous declaration specifies that the NUM symbol is a fullword signed integer. Any value consistent with a fullword plus a sign bit can be assigned to the symbol NUM. For example:

```
LET  NUM,=,65537
```

NUM can be an integer 1 larger than a halfword.

- If you do not specify a symbol, the attributes are determined as follows.
  - If the symbol is defined to the assembler as type C, the attribute is CHARACTER.
  - If the symbol is defined to the assembler as type F or H, the attribute is SIGNED.
  - If the symbol is defined with an EQU statement, the attribute is REGISTER.
  - All other types are given the attribute UNSIGNED including undefined symbols.
- Use this macro primarily for items (or usage) unique to an individual program. Also, when possible, do not specify the length so that changes in size do not require recoding.
- Use the REGISTER attribute only with variables declared with an equate or with variables not previously defined.
- The DCLREG macro provides a convenient method of declaring each of the general registers, R0–R15. This allows the registers to be used in subsequent IF, DO, CASE macros, and as operands to the LET macro. See “DCLREG–Declare General Registers” on page 96 for more information about the DCLREG macro.

Examples

The following example shows several DCL macro statements.

```
DCL     WORK0=8
DCL     R2,REGISTER,2
DCL     MY0CCT,UNSIGNED,2
DCL     MY1CCT,SIGNED,2
DCL     MY0TXT,CHARACTER,99
DCL     EBW020,CHARACTER,4
DCL     FRIEDT
DCL     LIST=YES
```

Related Information

“DCLREG–Declare General Registers” on page 96.

DCLREG–Declare General Registers

Use this macro to define all general registers with a name starting with the letter R. For example, general register 5 is defined as R5. The new register definitions can then be used with other TPF structured programming macros (SPMs).

Format

Entry Requirements

None.

Return Conditions

Control is returned to the next sequential instruction.

Programming Considerations

Coding a DCLREG macro is equivalent to coding the following DCL macro statements:

```
DCL   R0,REGISTER,0
DCL   R1,REGISTER,1
DCL   R2,REGISTER,2
DCL   R3,REGISTER,3
DCL   R4,REGISTER,4
DCL   R5,REGISTER,5
DCL   R6,REGISTER,6
DCL   R7,REGISTER,7
DCL   R8,REGISTER,8
DCL   R9,REGISTER,9
DCL   R10,REGISTER,10
DCL   R11,REGISTER,11
DCL   R12,REGISTER,12
DCL   R13,REGISTER,13
DCL   R14,REGISTER,14
DCL   R15,REGISTER,15
```

See “DCL–Declare” on page 93 for more information about the DCL macro.

Examples

The following defines all general registers with a name that starts with the letter R.

```
DCLREG
```

Related Information

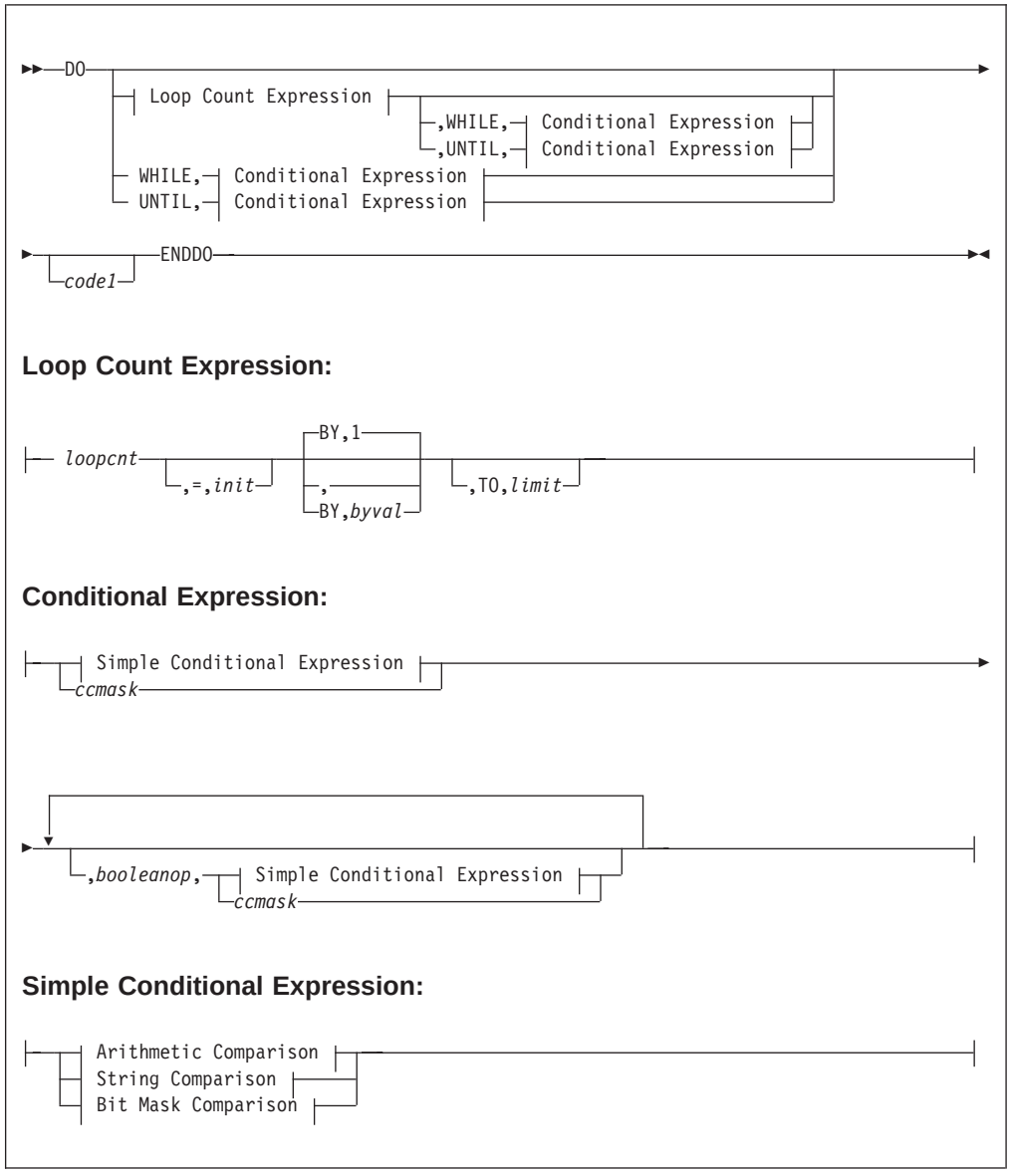
“DCL–Declare” on page 93.

DO Macro Group

Use this macro group to process specific code repeatedly based on a condition.

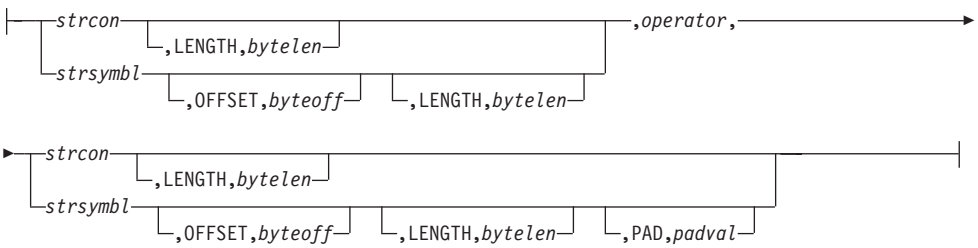
The DO macro group includes the following macros:
- DO
- ENDDO.

Format

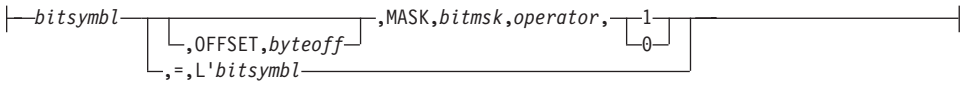


Arithmetic Comparison:

String Comparison:



Bit Mask Comparison:



DO
specifies the start of the DO structure. If you do not specify any parameters, an unconditional loop is generated; use the LEAVE or GOTO macro to avoid creating an infinite loop. See "LEAVE–Exit from a DO Loop" on page 111 and "GOTO–Branch Macro" on page 103 for more information about the LEAVE and GOTO macros.

loopcnt
is an arithmetic variable used to control the loop. An arithmetic variable is a symbol that represents a numeric value.

init
is an initial value for the loop counter (*loopcnt*). The initial value can be one of the following:

- Variable
- Constant
- An arithmetic expression.

If you do not specify an initial value, the TPF system assumes that the loop counter (*loopcnt*) is already initialized.

BY,*byval*
specifies a value that is used to increment or decrement the loop counter, where *byval* is a number or a symbol that represents a number.

If *byval* is preceded by a minus sign (−), the value is used to decrement the loop counter. For example,

```
BY,-,1
```

TO,*limit*
specifies when to end the loop, where *limit* is a number or a symbol that represents a number. The *limit* is compared to the loop counter, and when the value of *loopcnt* is greater than the value of *limit* the loop exits. If *byval* is preceded by a minus sign (−), the loop exits when *loopcnt* is less than *limit*.

If you specify a loop counter and do not specify a WHILE or UNTIL parameter, you must specify the TO limit.

WHILE
specifies the start of a DO WHILE loop based on a conditional expression. The loop is processed only if and while the conditional expression is true.

UNTIL
specifies the start of a DO UNTIL loop based on a conditional expression. The loop is processed only if and while the conditional expression is false.

ccmask
is any value that can be used as the mask in a branch on condition instruction. The SPMEQ equate macro defines some of these condition code masks.

booleanop
is one of the following Boolean connectors:
- AND
- OR.

arithval
is a number represented directly in numeric form or in symbolic form. This value can be one of the following:
- Variable
- Constant
- Result of an arithmetic expression.

operator
is one of the following relational operators:

Operator	Description
=	Equal
EQ	Equal
¬=	Not equal
NE	Not equal
<	Less than
LT	Less than
>	Greater than
GT	Greater than
<=	Less than or equal
LE	Less than or equal
>=	Greater than or equal
GE	Greater than or equal.

strcon
is a string constant, which can be one of the following:
- Character (C"HELLO")
- Hexadecimal (X'4040404040')
- Binary (B'10101010').

strsymbl
is a symbol that represents a character string. This symbol must be defined with a CHARACTER attribute. See “DCL–Declare” on page 93 for more information about defining the attribute for a symbol.

OFFSET,*byteoff*
specifies a substring, where *byteoff* is the distance (starting from zero) from the leftmost byte of the string. For example,

```
C"12345",OFFSET,3
```

resolves to a substring of C"45".

See the restrictions listed in the programming considerations.

LENGTH,*bytelen*
specifies a substring, where *bytelen* is the length of the string starting from the leftmost byte. For example,

```
C"12345",LENGTH,4
```

resolves to a substring of C"1234".

See the restrictions listed in the programming considerations.

PAD,*padval*
concatenates a character, *padval*, on the end of a string to fill out the string to the end of the declared length.

See the restrictions listed in the programming considerations.

bitsymbl
is a symbol that represents a value for the bit mask comparison. This symbol can be defined with any attribute, but only the leftmost byte is used in the comparison. See “DCL–Declare” on page 93 for more information about defining the attribute for a symbol.

L'*bitsymbl*
specifies the length of *bitsymbl* and resolves to the number of bytes of storage that *bitsymbl* represents.

MASK,*bitmsk*
specifies a bit mask, where *bitmsk* is a value that can be used as immediate data for a test under mask (TM) instruction. See *ESA/370 Principles of Operation* or *ESA/390 Principles of Operation* for more information about the TM instruction.

code1
is the code to process.

ENDDO
ends the loop processing.

Entry Requirements

None.

Return Conditions

- Control is returned to the next sequential instruction after the ENDDO macro statement unless another assembler instruction or macro passes control outside the DO structure.
- Any combination (or none) of the four work registers (default R0–R1, R14–R15) can be used by the DO macro. A message is generated for each work register used. The contents of each work register used are unknown. The work registers can be changed by coding the WORK0 and WORK2 parameters of the DCL macro. Each of these parameters must be the even register of an even-odd pair. The contents of all other registers are preserved across this macro call.

Programming Considerations

- The DO and ENDDO macros can only be used with the DO macro group.
- Each macro statement and assembler instruction must begin on a new line in the application.

- A section of code (represented by *code1*, and so on) can consist of any number of standard assembler instructions, including other SPMs or assembler macros.
- Because the SPMs are assembler language macros, all symbols used with the macros must be previously defined to the assembler. In addition, for the TPF SPMs, you must declare the attributes of the symbols using the DCL macro.
- If you specify a DO macro with no parameters, the loop will process forever unless you code a LEAVE macro or an explicit branch instruction. See "LEAVE–Exit from a DO Loop" on page 111 for more information about the LEAVE macro.
- If you specify both the OFFSET and LENGTH parameters, you must code the OFFSET parameter first.
- If you specify the OFFSET and PAD parameters with the second operand, you must specify the LENGTH parameter on the second operand as well.
- The DO loop is processed as follows:
  1. The loop counter begins at its initial value.
  2. The application program processes through the loop.
  3. The increment is added to the control variable.
  4. A test is made to ensure the result is below the limit.
  5. If below the limit, the application program processes through the loop again; otherwise, the program branches to the statement following the ENDDO.
- If symbols referred to by *loopcnt*, *byval*, or *limit* are changed during loop processing, the results cannot be predicted.

  Symbols in the conditional expression are evaluated at the beginning of each repetition and can be changed in the DO group to end the loop.
- The most efficient code is produced when variables have the REGISTER attribute. (See "DCL–Declare" on page 93 for more information about specifying the attribute for a symbol.) In particular, use a register as the control variable for the following two special cases:
  1. DO *variable*,BY,−,1,TO,1
  2. DO *variable*,BY,*x*,TO,*y*

  Where *y* is an odd-numbered register and *x* is the same as *y* or is the next lower register.

  Case 1 will use a BCT instruction to control the loop, and case 2 will use a BXLE instruction if the control variable is a register.

Examples

- The following example shows a simple DO loop.

```
LET ANS1,=,0
DO NUM1,=,1,TO,10
   LET ANS1,=,ANS1,+,1
ENDDO
```

- The following example shows a simple DO WHILE loop.

```
LET ANS1,=,0
DO NUM1,WHILE,ANS1,LT,10
   LET ANS1,=,ANS1,+,1
ENDDO
```

- The following example shows a simple DO UNTIL loop.

```
LET ANS1,=,0
DO NUM1,UNTIL,ANS1,GE,10
   LET ANS1,=,ANS1,+,1
ENDDO
```

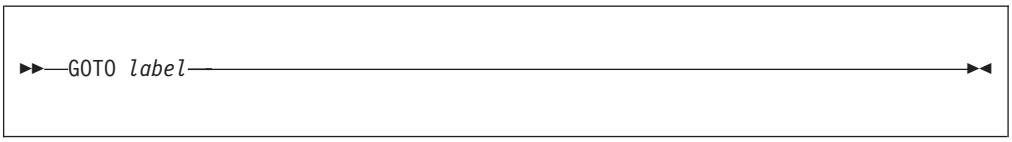
Related Information

- “DCL–Declare” on page 93
- “IF Macro Group” on page 104
- “LEAVE–Exit from a DO Loop” on page 111

GOTO–Branch Macro

Use this macro to pass control to another section of code.

Format



label
is a symbolic label where control will be passed.

Entry Requirements

None.

Return Conditions

Control is passed to the label specified on the GOTO macro.

Programming Considerations

The *label* cannot be coded between the GOTO macro and the next structured programming macro at the same level of nesting.

Examples

The following example causes a branch to label ENDLOOP.

```
GOTO ENDLOOP
```

Related Information

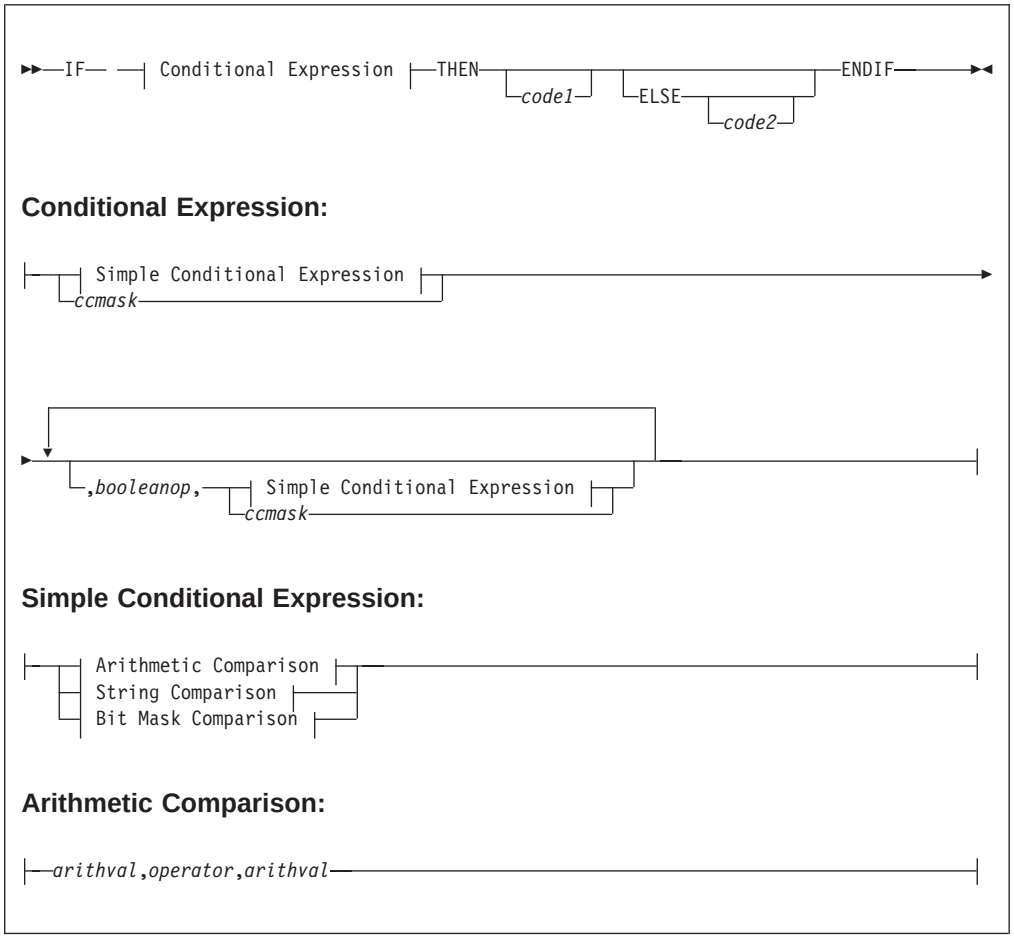
None.

IF Macro Group

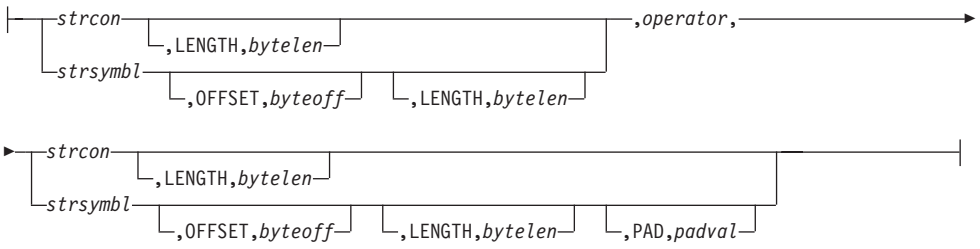
Use this macro group to define code process specific code based on a set of conditions. The IF macro group includes the following macros:

- IF
- THEN
- ELSE
- ENDIF.

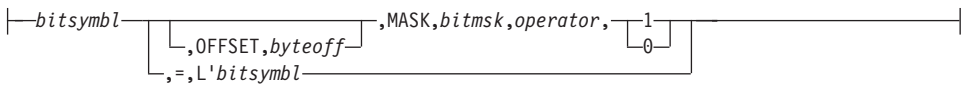
Format



String Comparison:



Bit Mask Comparison:



IF specifies the start of the IF structure based on a conditional expression.

ccmask
is any value that can be used as the mask in a branch on condition instruction. The SPMEQ equate macro defines some of these condition code masks.

booleanop
is one of the following Boolean connectors:
- AND
- OR.

arithval
is a number represented directly in numeric form or in symbolic form. This value can be one of the following:
- Variable
- Constant
- Result of an arithmetic expression.

operator
is one of the following relational operators:

Operator	Description
=	Equal
EQ	Equal
¬=	Not equal
NE	Not equal
<	Less than
LT	Less than
>	Greater than
GT	Greater than
<=	Less than or equal
LE	Less than or equal
>=	Greater than or equal
GE	Greater than or equal.

strcon
is a string constant, which can be one of the following:
- Character (C"HELLO")
- Hexadecimal (X'4040404040')
- Binary (B'10101010').

strsymbl
is a symbol that represents a character string. This symbol must be defined with a CHARACTER attribute. See “DCL–Declare” on page 93 for more information about defining the attribute for a symbol.

OFFSET,*byteoff*
specifies a substring, where *byteoff* is the distance (starting from zero) from the leftmost byte of the string. For example,

```
C"12345",OFFSET,3
```

resolves to a substring of C"45".

See the restrictions listed in the programming considerations.

LENGTH,*bytelen*
specifies a substring, where *bytelen* is the length of the string starting from the leftmost byte. For example,

```
C"12345",LENGTH,4
```

resolves to a substring of C"1234".

See the restrictions listed in the programming considerations.

PAD,*padval*
concatenates a character, *padval*, on the end of a string to fill out the string to the end of the declared length.

See the restrictions listed in the programming considerations.

bitsymbl
is a symbol that represents a value for the bit mask comparison. This symbol can be defined with any attribute, but only the leftmost byte is used in the comparison. See “DCL–Declare” on page 93 for more information about defining the attribute for a symbol.

L'*bitsymbl*
specifies the length of *bitsymbl* and resolves to the number of bytes of storage that *bitsymbl* represents.

MASK,*bitmsk*
specifies a bit mask, where *bitmsk* is a value that can be used as immediate data for a test under mask (TM) instruction. See *ESA/370 Principles of Operation* or *ESA/390 Principles of Operation* for more information about the TM instruction.

THEN
specifies the start of the code to process when the conditional expression is true.

code1
is the code to process when the conditional expression is true.

ELSE
specifies the start of the code to process when the conditional expression is false.

code2
is the code to process when the conditional expression is false.

ENDIF
ends the IF structure.

Entry Requirements

None.

Return Conditions

- Control is returned to the next sequential instruction after the ENDIF macro statement unless another assembler instruction or macro passes control outside of the IF structure.
- Any combination (or none) of the four work registers (default R0–R1, R14–R15) can be used by the IF macro. A message is generated for each work register used. The contents of each work register used are unknown. The work registers can be changed by coding the WORK0 and WORK2 parameters of the DCL macro. Each of these parameters must be the even register of an even-odd pair. The contents of all other registers are preserved across this macro call.
- If the expression is true, control is returned to the next sequential instruction (the THEN macro). Otherwise, control is given to the matching ELSE macro. If the ELSE macro is not used, control is passed to the matching ENDIF macro.

Programming Considerations

- The IF, THEN, ELSE, and ENDIF macros can only be used with the IF macro group.
- Each macro statement and assembler instruction must begin on a new line in the application.

  Note: You can code the THEN macro on the same line as the IF expression, but you must separate the expression and the THEN macro with a comma (,).
- A section of code (represented by *code1*, and so on) can consist of any number of standard assembler instructions, including other SPMs or assembler macros.
- Because the SPMs are assembler language macros, all symbols used with the macros must be previously defined to the assembler. In addition, for the TPF SPMs, you must declare the attributes of the symbols using the DCL macro.
- All operations are performed left to right except when a sublist (items in parentheses) is encountered. All items within a sublist are evaluated before applying the preceding operator to the sublist.
- If you specify both the OFFSET and LENGTH parameters, you must code the OFFSET parameter first.
- If you specify the OFFSET and PAD parameters with the second operand, you must specify the LENGTH parameter on the second operand as well.
- The conditional expression is evaluated as follows:
  1. If you code only one operand, or if the first operand is not a sublist but is followed by THEN, AND, or OR, the first operand is treated as a condition code mask. The hardware condition code is tested against the mask. If a branch on condition instruction with this mask would have branched, the condition is evaluated as true. Otherwise, it is evaluated as false (see step 6 on page 108).
  2. If the first (next) item is a sublist, the key parameter following the sublist is checked. If it is THEN, AND, or OR, the sublist is evaluated as a conditional expression as in step 3 on page 108 through step 6 on page 108. Otherwise it is evaluated as an arithmetic expression and is used as the left side of an arithmetic comparison. Note that the arithmetic expression cannot contain sublists.

3. A conditional expression is treated as a bit mask comparison if the MASK key parameter appears. The bits specified by the mask are tested for 0 or 1 as indicated by the right side value.
4. A conditional expression is treated as an arithmetic comparison if the left side does not have the CHARACTER attribute. (See “DCL–Declare” on page 93 for information about defining the attribute for a symbol.) The left and right side values are first converted to a 32-bit binary number. Symbols with the UNSIGNED attribute are extended on the left with zeros. Otherwise, the high-order bit is propagated and used as the sign of the number.

   If either side is a sublist (enclosed in parentheses), it is first evaluated as an arithmetic expression and the result is placed in one of the work registers so that the comparison can be performed.

   The comparison is then performed. The comparison is logical if the left side symbol is UNSIGNED and not enclosed in parentheses. Otherwise, the comparison is algebraic; that is, the high-order bit is treated as the sign of each number.
5. A string comparison is performed when the left side symbol has the CHARACTER attribute. The right side must be a string constant or a symbol with the CHARACTER attribute. The comparison is performed left to right using the length of the shortest field unless the PAD key parameter is used. In this case, the shorter string is extended conceptually with bytes containing the pad value to the length of the longer string.
6. If the comparison results match the relational operator the conditional expression is true. Otherwise it is false. If the conditional expression is true and is followed by OR, the rest of the IF expression is skipped and the entire expression is evaluated as true. (If the conditional expression is in a sublist, the rest of the sublist is skipped and the sublist is evaluated as true).

   If the conditional expression is false and is followed by AND, the remainder of the expression (or sublist) is skipped and evaluated as false.

   Otherwise, the next expression is evaluated as in item 2.

   If and when the last (or only) relational expression is evaluated, it determines the evaluation of the whole expression (or sublist).
7. The end of the IF expression is indicated by a null operand or the THEN key parameter in an operator position. The THEN key parameter causes the THEN macro to be processed. Otherwise, the IF macro must be immediately followed by a THEN macro.

Examples

The following examples assume that the attributes for the referenced symbols have been defined with following DCL macro statements:

```
DCL   EOFSW,UNSIGNED,1
DCL   I,SIGNED,4
DCL   X,SIGNED,4
DCL   Y,SIGNED,4
DCL   Z,SIGNED,2
DCL   AMOTXT,CHARACTER,99
DCL   ARG,CHARACTER,4
DCL   R3,REGISTER,3
IF    EOFSW,=,0
```

- In the following example, a specific section of code is processed if the expression ((EOFSW=1) OR (I>(X+Y))) AND (Z<3) is true.

```
         IF     EOFSW,=,1,OR,I,GT,(X,+,Y),AND,Z,LT,3
         THEN
           :
*   Code to process
           :
         ENDIF
```

- In the following example, a specific section of code is processed if AM0TXT=C"ZLXXX". If AM0TXT does not equal C"ZLXXX", the section of code following the ELSE macro is processed.

```
         IF    AM0TXT,=,C'ZLXXX'
         THEN
           :
*   Code to process
           :
         ELSE
           :
*   Code to process
           :
         ENDIF
```

- In the following example, the expression is evaluated with the OFFSET, LENGTH, and PAD parameters.

```
         IF    AM0TXT,=,ARG,OFFSET,(X,+,Y),LENGTH,3,PAD,C' '
         THEN
           :
*   Code to process
           :
         ENDIF
```

- The following example shows a bit mask comparison. If the bits set on in mask byte SLSTRCPL do not match those in test byte SLSTLC3, the test is true and the code following the THEN macro is processed. If any bits in SLSTRCPL and SLSTLC3 do match, the test is false and the code following the ELSE macro is processed.

```
         IF    SLSTLC3,MASK,SLSTRCPL,=,0
         THEN
           :
*   Code to process
           :
         ELSE
           :
*   Code to process
           :
         ENDIF
```

- The following example shows a test that uses a number.

```
BUSY     EQU   2
         IF    BUSY,THEN
           :
*   Code to process
           :
         ENDIF
```

- The following example shows the use of the length attribute in a bit mask comparison.

```
OPTION_1   DS   0XL1
OPTION_2   DS   0XL2
FLAG_BYTE  DS   XL1

*   Code to process

         IF FLAG_BYTE,MASK,L'OPTION_1,EQ,1
         THEN
           :
*   Code to process
           :
```

```
          ENDIF
            :
*   Code to process
            :
          IF FLAG_BYTE,MASK,L'OPTION_2,EQ,1
          THEN
            :
*   Code to process
            :
          ENDIF
```

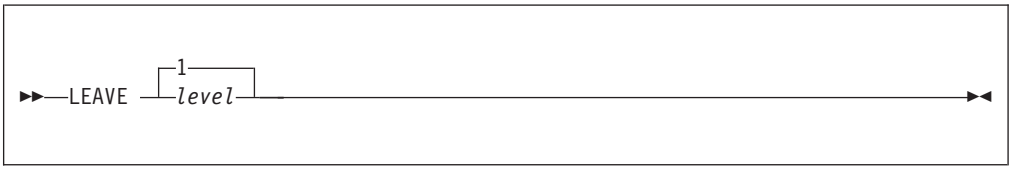
Related Information

- “DCL–Declare” on page 93
- “DO Macro Group” on page 97
- “LEAVE–Exit from a DO Loop” on page 111.

LEAVE–Exit from a DO Loop

Use this macro to exit a DO loop when the condition being tested by the corresponding CASE, IF, or SELECT macro is true.

Format



level
is a self-defining term or SETA assembler variable equal to the number of DO loop nesting levels to leave. For example, if a DO loop contains a nested DO loop, coding LEAVE 2 in the inner DO loop causes processing to branch to the outer ENDDO macro, which ends the outer DO loop. If you do not specify *level*, the default is 1; that is, a branch is taken to the end of the innermost DO loop that contains the LEAVE macro.

Entry Requirements

You must code this macro between a DO and ENDDO macro group and after a CASE, IF-THEN, or SELECT macro statement.

Return Conditions

When the condition being tested for is true, LEAVE causes processing to continue with the next sequential instruction following the ENDDO statement of the current DO-ENDDO block, or as many levels of DO-ENDDO blocks as specified by the level parameter.

Programming Considerations

None.

Examples

This example sets up a DO loop to add 1 to the contents of R1 10 times. When the value of control variable LPCNT exceeds 8, the IF statement is true and the LEAVE macro is processed. This causes processing to continue with R1 equal to 8 beginning after the ENDDO statement.

```
LA R1,0
DO LPCNT,=,1,TO,10
   IF LPCNT,GT,8
     THEN
        LEAVE ,
   ENDIF
LA R1,1(,R1)
ENDDO
```

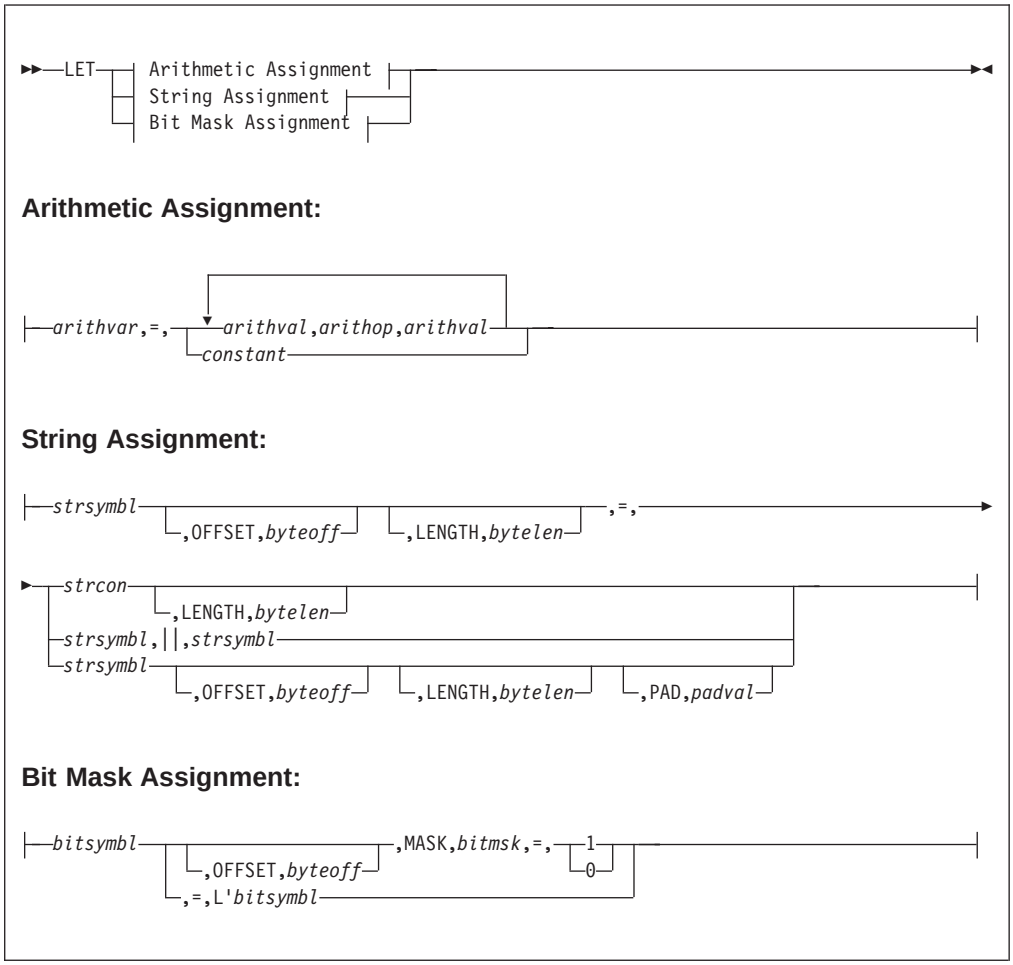
Related Information

- “DO Macro Group” on page 97
- “IF Macro Group” on page 104.

LET–Assignment

Use this macro to assign values to symbols declared with the DCL macro. Symbols can be assigned arithmetic, string, or bit values. These symbols can be used in other structured programming macro (SPM) or in mainline code.

Format



arithvar
is a symbolic name to which the arithmetic value will be assigned. For example:

```
LET NUM,=,4
```

arithval
is a number represented directly in numeric form or in symbolic form. This value can be one of the following:

- Variable
- Constant
- Result of an arithmetic expression.

arithop
is one of the following arithmetic operators:

Operator	Description
+	Addition
–	Subtraction
*	Multiplication

/ Integer division
// Remainder.

constant
is an arithmetic constant that will be assigned to the symbol specified by *arithvar*. The value must be an integer and must be appropriate for the size of the symbol to which this value will be assigned (*arithvar*). For example, if the number you specify for *constant* is greater than 65536, the symbol you specify for *arithvar* must be declared to be a fullword. See “DCL–Declare” on page 93 for more information about declaring the attributes for a symbol.

strcon
is a string constant, which can be one of the following:
- Character (C"HELLO")
- Hexadecimal (X'4040404040')
- Binary (B'10101010').

strsymbl
is a symbol that represents a character string. This symbol must be defined with a CHARACTER attribute. See “DCL–Declare” on page 93 for more information about defining the attribute for a symbol.

|| concatenates two or more strings into one string. For example,

```
LET PART1,=,C"ABC"
LET PART2,=,C"123"   LET WHOLE,=,PART1,||,PART2
```

In this example, WHOLE will have the value C"ABC123".

OFFSET,*byteoff*
specifies a substring, where *byteoff* is the distance (starting from zero) from the leftmost byte of the string. For example,

```
C"12345",OFFSET,3
```

resolves to a substring of C"45".

See the restrictions listed in the programming considerations.

LENGTH,*bytelen*
specifies a substring, where *bytelen* is the length of the string starting from the leftmost byte. For example,

```
C"12345",LENGTH,4
```

resolves to a substring of C"1234".

See the restrictions listed in the programming considerations.

PAD,*padval*
concatenates a character, *padval*, on the end of a string to fill out the string to the end of the declared length.

See the restrictions listed in the programming considerations.

bitsymbl
is a symbol that represents a value for the bit mask assignment. In a bit mask assignment, either the leftmost byte or the one byte specified by the OFFSET parameter is used.

L'*bitsymbl*
specifies the length of *bitsymbl* and resolves to the number of bytes of storage that *bitsymbl* represents.

MASK,*bitmsk*
specifies a bit mask, where *bitmsk* is a value that specifies the portions of the byte affected by the bit that is being assigned. For example:

```
LET SYMB,=,C"ABC"
LET SYMB,MASK,B'00101000',=,1
```

In this example, the symbol SYMB will contain the value ZBC. This is determined as follows:

```
    EBCDIC A: B'1100 0001'
     mask on: B'0010 1000'
  set to ones: ------------
   results in: B'1110 1001' or EBCDIC Z
```

Entry Requirements

None.

Return Conditions

- Control is returned to the next sequential instruction.
- Any combination (or none) of the four work registers (default R0–R1, R14–R15) can be used by the LET macro. A message is generated for each work register used. The contents of each work register used are unknown. The work registers can be changed by coding the WORK0 and WORK2 parameters of the DCL macro. Each of these parameters must be the even register of an even-odd pair. The contents of all other registers are preserved across this macro call.

Programming Considerations

- Because the SPMs are assembler language macros, all symbols used with the macros must be previously defined to the assembler. In addition, for the TPF SPMs, you must declare the attributes of the symbols using the DCL macro.
- A LET macro expression is limited to 13 items. That is, in addition to assigning the symbol and the equal sign (=), 11 additional symbols, numbers, or operators can be contained in any LET macro. If expressions are longer than 13 items, they must be broken up.
- You can concatenate several strings on a single LET macro statement. However, do not specify concatenation with the OFFSET or LENGTH parameters.
- If you specify both the OFFSET and LENGTH parameters, you must code the OFFSET parameter first.

Examples

- The following example shows the assignment of a constant arithmetic value.

```
*   Declare a full word, signed integer called TEST
TEST     EQU   EBW000,4                    Reentrant for TEST  DS   F
         DCL   TEST,SIGNED,4
*
*   Assign variable TEST the decimal value 255
         LET   TEST,=,255
```

- The following example shows the assignment of a variable arithmetic value.

```
*   Declare a several arithmetic variables
*
SUMM     EQU   EBW000,4                    Reentrant for SUMM  DS   F
         DCL   SUMM,SIGNED,4

MULT     EQU   EBW004,4                    Reentrant for MULT  DS   F
         DCL   MULT,SIGNED,4
```

```
ANSWER   EQU   EBW008,4                     Reentrant for ANSWER DS  F
         DCL ANSWER,SIGNED,4
           :
*   Assign arithmetic values
         LET   SUMM,=,2,+,4,+,6
         LET   MULT,=,4,*,3
*
         LET   ANSWER,=,SUMM,/,MULT
```

- The following example shows the assignment of string values.

```
*   Declare a string 80 bytes long called OUT
OUT      EQU   EBW000,80                   Reentrant for OUT  DS  CL80
         DCL   OUT,CHARACTER,80
           :
*   Declare a string 20 bytes long called NAME
NAME     EQU   EBX000,20                    Reentrant for NAME  DS  CL20
         DCL   NAME,CHARACTER,20
           :
*   Declare a string 40 bytes long called ADDRESS
ADDRESS  EQU   EBX020,40                Reentrant for ADDRESS DS  CL40
         DCL   ADDRESS,CHARACTER,40
           :
*   Declare a string 20 bytes long called PHONE
PHONE    EQU   EBX060,20                  Reentrant for PHONE  DS  CL20
         DCL   PHONE,CHARACTER,20
           :
*   Assign variable NAME a constant string value
         LET   NAME,=,C'Robert Cohen',PAD,' '
           :
*   Assign variable ADDRESS a constant string value
         LET   ADDRESS,=,C'40 Apple Ridge Road $ Danbury, Ct 06810'
           :
*   Assign variable PHONE a constant string value
         LET   PHONE,=,C'  (203) 790-2000',PAD,' '
           :
*   Assign variable OUT the concatenations of NAME, ADDRESS, and PHONE
         LET OUT,=,NAME,||,ADDRESS,||,PHONE
```

- The following is another example of assigning string values.

```
*   Declare some strings

PART1    EQU   EBW000,5                   Reentrant for PART1  DS  CL5
PART2    EQU   EBW005,5                   Reentrant for PART2  DS  CL5
MIDL     EQU   EBW010,20                  Reentrant for MIDL   DS  CL20
WHOLE    EQU   EBX000,40                  Reentrant for WHOLE  DS  CL40
           :
         DCL   PART1,CHARACTER,5
         DCL   PART2,CHARACTER,5
         DCL   MIDL,CHARACTER,20
         DCL   WHOLE,CHARACTER,40
           :
*   Assign variable WHOLE a constant string of 40 characters
         LET WHOLE,=,C'ABCDEFGHIJ1234567890KLMNOPQRST0987654321'
           :
*   Assign variable PART1 the first five characters of WHOLE
         LET PART1,=,WHOLE,LENGTH,5
*   PART1 is 'ABCDE'
           :
*   Assign variable PART2 the last five characters of WHOLE
         LET PART2,=,WHOLE,OFFSET,35
*   PART2 is '54321'
           :
*   Assign variable MIDL 10 characters from WHOLE starting 15 characters
*   from the beginning of WHOLE.  Pad MIDL with blanks
         LET MIDL,=,WHOLE,OFFSET,15,LENGTH,10,PAD,' '
*   MIDL is '67890KLMNObbbbbbbbbb'  where 'b' represents a 'blank'
```

- The following example shows an assignment using a bit mask. This example sets the 4 most significant bits of TEST to binary ones. The 4 least significant bits of TEST are not affected.

```
TEST     DS    B
         DCL   TEST,CHARACTER,1
         LET   TEST,MASK,B'11110000',=,1
```

- The following example shows how a bit mask assignment is used to convert uppercase characters to lowercase characters in a DO loop.

  In this example, the symbol INDX is declared as an unsigned variable and the string variable STR is initialed as THIS IS A TEST with a length of 14. The DO loop initializes the value of INDX to be 1. The LET assignment in the DO loop is coded in an IF structure to ensure that the byte specified by the value of INDX is not blank. When not blank, the LET assignment takes the value of string STR, offsets the byte specified by the value of INDX, and casts a mask over that byte. The mask isolates the high-order bit of the byte that is being specified. The bit is assigned value 0. The remaining bits of the byte are unaffected (because the mask has zeros in their places). The original value of string STR is changed by this byte change. The value of INDX is incremented by 1 at the top of the DO loop, and the loop proceeds with INDX being 2, 3, until 14. Each value of INDX specifies a different byte in string STR.

```
DCL   INDX,UNSIGNED
DCL   STR,CHARACTER,14
 :
LET   STR,=,C'THIS IS A TEST'
DO    INDX,=,1,TO,14
  IF    STR,OFFSET,INDX,NE,C' '
     THEN
        LET STR,OFFSET,INDX,MASK,B'10000000',=,0
  ENDIF
ENDDO
```

Related Information

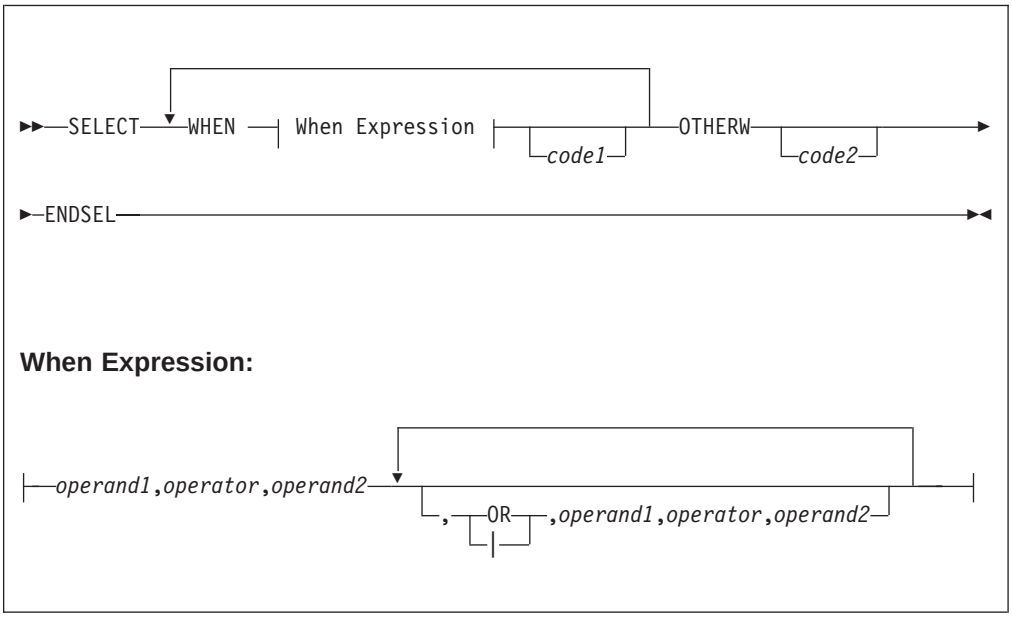
"DCL–Declare" on page 93.

SELECT Macro Group

Use this macro group to control the choice among a number of different code paths by a logical comparison. The SELECT macro group is similar to the CASE macro group except that instead of using an arithmetic expression to select an alternative clause, SELECT conditions are successively evaluated until an alternative can be selected.

The SELECT macro group includes the following macros:
- SELECT
- WHEN
- OTHERW
- ENDSEL.

Format



SELECT
specifies the start of the SELECT structure.

WHEN
specifies the start of a selection.

operand1
is a symbol, a register enclosed in parentheses, or a literal that can be used as the first operand on a compare or test under mask (TM) instruction.

operand2
is a symbol or a register enclosed in parentheses that can be used as the second operand on a compare or test under mask (TM) instruction.

operator
is one of the following relational operators:

Operator	Description
EQ	Equal
=	Equal
NE	Not equal
¬=	Not equal

LE	Less than or equal
<=	Less than or equal
LT	Less than
<	Less than
GE	Greater than or equal
>=	Greater than or equal
GT	Greater than
>	Greater than
Z	Zeroes (or OFF)
O	Ones (or ON)
M	Mixed zeros and ones
NZ	Not zeros
NO	Not ones
NM	Not mixed.

OR
allows additional expressions to be evaluated on one WHEN statement.

| allows additional expressions to be evaluated on one WHEN statement.

code1
is the code to process when the associated WHEN statement is true.

OTHERW
specifies the start of the code to process when all the previous WHEN statements are false.

code2
is the code to process when the previous WHEN statements are false.

ENDSEL
ends the SELECT structure.

Entry Requirements

None.

Return Conditions

- If the conditional expression associated with a WHEN clause is true, the code following the WHEN clause is processed until the next WHEN or OTHERW clause in that SELECT macro is found.
- If the conditional expression is false in a particular WHEN macro statement, control is passed to the next sequential WHEN macro or the OTHERW macro if there are no more WHEN macros in the current SELECT structure.
- After the instructions following a particular WHEN or OTHERW macro statement are processed, control is passed to the ENDSEL macro.
- If the conditional expression for more than one WHEN is true, only the code following the first true WHEN is processed.
- The contents of all the user registers are preserved across this macro call.

Programming Considerations

- The SELECT, WHEN, OTHERW, and ENDSEL macros can only be used with the SELECT macro group.
- Each macro statement and assembler instruction must begin on a new line in the application.
- A section of code (represented by *code1*, and so on) can consist of any number of standard assembler instructions, including other SPMs or assembler macros.

- Because the SPMs are assembler language macros, all symbols used with the macros must be previously defined to the assembler. In addition, for the TPF SPMs, you must declare the attributes of the symbols using the DCL macro.
- You can specify a maximum of 39 operands with the WHEN macro.
- If you specify the OR parameter, if any of the expressions separated by the OR parameter is true, the whole WHEN macro statement is true.
- The SELECT and ENDSEL sequence of macros can be nested up to a limit of eight.

Examples

The following is an example of a SELECT macro group.

```
         SELECT
            WHEN  EBW000,EQ,OPTION1
            :
*   Code to process
            :
            WHEN  EBW000,EQ,OPTION2,OR,EBW000,EQ,OPTION3
            :
*   Code to process
            :
            OTHERW
            :
*   Code to process
            :
         ENDSEL
```

Related Information

- “DCL–Declare” on page 93
- “CASE Macro Group” on page 90.

SET–Flag or Switch Assignment

Use this macro to turn on 1 or more bits in a byte or to replace the value of an entire byte. A single macro parameter determines both the type of assignment and the assignment operands.

Format

setsymbol
is an equate or assembler label from which the following is determined:

- The address of a byte in storage. The *setsymbol* itself (that is, its value as defined to the assembler) determines the address of the byte in storage.
- An immediate value. The length attribute of the *setsymbol* determines the immediate value.
- How the immediate value affects the byte in storage. The type attribute of the *setsymbol* determines whether the immediate value is:
  – Assigned to the byte in storage
  – Used as a mask for turning on selected bits within the byte in storage.

Entry Requirements

None.

Return Conditions

- Control is returned to the next sequential instruction.
- The contents of all registers are preserved across this macro call.

Programming Considerations

- Any number of *setsymbol*s can be coded for a single SET macro statement.
- If a *setsymbol*-type attribute is X, its length attribute is taken as a bit-string and ORed with the contents of the byte. Otherwise, the entire content of the byte is replaced by the length attribute.
- There are two ways to define a *setsymbol*:
  – Define a label using DS. Set the location counter to the displacement of the byte to be set, and code the DS operand to define the desired length and type attributes.
  – Define an equated symbol using EQU with all 3 operands.
    1. Code the first EQU operand as the symbolic address of the byte to be set.
    2. Code the second operand as the value to be set.
    3. Code the third operand as C"X" to generate an or immediate (OI) instruction, or as any value other than C"X" to generate a move immediate (MVI) instruction.

       The third EQU operand must be explicitly coded as C"X" to OR on selected bits of a byte because the default type for equates is U.

Examples

- The following example shows the difference between setting a symbol that is defined as type X and setting a symbol that is defined as something other than type X (such as U).

```
*****************************
* TYPE = 'X' VS. TYPE ^= 'X' *
* BITWISE-OR VS. ASSIGNMENT  *
*****************************
*
*  A_BIT_FLAG = X'08' (Bit 4 is on)
*  A_BYTE_FLAG = X'BC'
         SET   A_BIT_FLAG,A_BYTE_FLAG
+        OI    A_BIT_FLAG,L'A_BIT_FLAG
+        MVI   A_BYTE_FLAG,L'A_BYTE_FLAG
               :
               :
CONTROL_BYTE   DS    XL1                      STORAGE FOR BIT FLAGS
FLAG_BYTE      DS    XL1                      STORAGE FOR BYTE FLAG
A_BIT_FLAG     EQU   CONTROL_BYTE,X'08',C'X'  TYPE IS 'X'
A_BYTE_FLAG    EQU   FLAG_BYTE,X'BC'          TYPE IS 'U'
```

- In the following example, the set symbols are defined using a DS statement.

```
*********************************
* DEFINING SETSYMBOLS USING DS *
*********************************
         SET   TAG_ASGN2                    TAG_BYTE = X'02'
+        MVI   TAG_ASGN2,L'TAG_ASGN2
               :
               :
TAG_BYTE       DS    X                     STORAGE FOR TAGS
               ORG   TAG_BYTE              RESET LOCATION COUNTER
TAG_ASGN1      DS    0CL1                  DEFINE 3 ASSIGNMENTS
TAG_ASGN2      DS    0CL2
TAG_ASGN3      DS    0CL3
               ORG
```

- In the following example, the set symbols are defined using an EQU statement.

```
*********************************
* DEFINING SETSYMBOLS USING EQU *
*********************************
         SET   SWITCH1                      BIT 7 OF CNTRL_BYTE = B'1'
+        OI    SWITCH1,L'SWITCH1
               :
               :
CNTRL_BYTE     DS    X                     STORAGE FOR SWITCHES
SWITCH1        EQU   CNTRL_BYTE,B'00000001',C'X'
SWITCH2        EQU   CNTRL_BYTE,B'00000010',C'X'
SWITCH3        EQU   CNTRL_BYTE,B'00000100',C'X'
```

Related Information

None.

Index

Special characters

A

B

C

D

E

F

G

I

L

M

N

P

R

S

T

W

File Number: S370/30XX-40
Program Number: 5706-196
5748-T14

Printed in U.S.A.

SH31-0183-04