

High Level Assembler:
Toolkit Feature Technical Overview
SHARE 101 (August 2003), Session 8166

August 13, 2003

John R. Ehrman
ehrman@us.ibm.com or ehrman@vnet.ibm.com

International Business Machines Corporation
Silicon Valley (nee Santa Teresa) Laboratory
555 Bailey Avenue
San Jose, California 95141 USA

Synopsis:

This document provides an overview of the IBM High Level Assembler for MVS & VM & VSE Toolkit Feature and shows how its components can be used at all stages of program development and deployment.

HLASM documentation, presentation materials, and demonstration and trial versions of some Toolkit components are available on the HLASM web site:

<http://www.ibm.com/software/ad/hlasm/>

The examples in this document are for purposes of illustration only, and no warranty of correctness or applicability is implied or expressed.

Permission is granted to SHARE Incorporated to publish this material in the proceedings of the SHARE 101 (August 2003). IBM retains the right to publish this material elsewhere.

© IBM Corporation 1995, 2003. All rights reserved.

Notice

© IBM Corporation 1995, 2003. All rights reserved. Note to U.S. Government Users: Documentation subject to restricted rights. Use, duplication, or disclosure is subject to restrictions set forth in GSA ADP Schedule Contract with IBM Corp.

Copyright Notices and Trademarks

Note to U.S. Government Users: Documentation subject to restricted rights. Use, duplication, or disclosure is subject to restrictions set forth in GSA ADP Schedule Contract with IBM Corp.

The following terms, denoted by an asterisk (*) in this publication, are trademarks or registered trademarks of the International Business Machines Corporation in the United States and/or other countries:

IBM	ESA	System/370	System/370/390
System/390	MVS/ESA	OS/390	VM/ESA
VSE/ESA	VSE	z/OS	z/VM
z/Architecture	zSeries	OS/2	OS/2 Warp
DFSMS			

The following are trademarks or registered trademarks of other corporations:

Windows 95 Windows 98 Windows 2000 Windows NT Windows XP

Publications, Collection Kits, Web Sites

The currently available product publications for High Level Assembler for MVS & VM & VSE are:

- High Level Assembler for MVS & VM & VSE *Language Reference*, SC26-4940
- High Level Assembler for MVS & VM & VSE *Programmer's Guide*, SC26-4941
- High Level Assembler for MVS & VM & VSE *General Information*, GC26-4943
- High Level Assembler for MVS & VM & VSE *Licensed Program Specifications*, GC26-4944
- High Level Assembler for MVS & VM & VSE *Installation and Customization Guide*, SC26-3494

- High Level Assembler for MVS & VM & VSE *Toolkit Feature Interactive Debug Facility User's Guide*, GC26-8709
- High Level Assembler for MVS & VM & VSE *Toolkit Feature User's Guide*, GC26-8710
- High Level Assembler for MVS & VM & VSE *Toolkit Feature Installation and Customization Guide*, GC26-8711
- High Level Assembler for MVS & VM & VSE *Toolkit Feature Interactive Debug Facility Reference Summary*, GC26-8712

- High Level Assembler for MVS & VM & VSE *Release 2 Presentation Guide*, SG24-3910

Soft-copy High Level Assembler for MVS & VM & VSE publications are available on the following *IBM Online Library Omnibus Edition* Compact Disks:

- *VSE Collection*, SK2T-0060
- *MVS Collection*, SK2T-0710
- *Transaction Processing and Data Collection*, SK2T-0730
- *VM Collection*, SK2T-2067
- *OS/390 Collection*, SK2T-6700 (BookManager), SK2T-6718 (PDF)

HLASM publications are available online at the HLASM web site:

<http://www.ibm.com/software/ad/hlasm/>

Formatted 09 Jul 03, 1245.

High Level Assembler Toolkit Feature

- Optional priced feature of High Level Assembler for MVS & VM & VSE
- Enhances productivity by providing six powerful tools:
 1. A flexible **Disassembler**
 - Creates symbolic Assembler Language source from object code
 2. A powerful Source **Cross-Reference Facility**
 - Analyzes code, summarizes symbol and macro use, locates specific tokens
 3. A workstation-based **Program Understanding Tool**
 - Provides graphic displays of control flow within and among programs
 4. A powerful and sophisticated **Interactive Debug Facility (IDF)**
 - Supports a rich set of diagnostic and display facilities and commands
 5. A complete set of **Structured Programming Macros**
 - Do, Do-While, Do-Until, If-Then-Else, Search, Case, Select, etc.
 6. A versatile **File Comparison Utility ("Enhanced SuperC")**
 - Includes special date-handling capabilities
- A comprehensive tool set for Assembler Language applications

HLASM Toolkit Feature

© IBM Corporation 1995, 2003. All rights reserved.

1

High Level Assembler Toolkit Feature

The High Level Assembler Toolkit Feature is an optional, separately priced feature of IBM High Level Assembler. It provides a powerful and flexible set of six tools to improve application recovery and development, and to assist in program preparation, analysis, debugging, and maintenance on z/OS*, z/VM*, OS/390*, MVS/ESA*, VM/ESA*, and VSE/ESA* systems. These productivity-enhancing tools are:

- **Disassembler**, a tool which converts binary machine language to Assembler Language source statements. It helps you understand programs in executable or object format, and enables recovery of lost source code.
- **Cross-Reference Facility**, a flexible source-code analysis and cross-referencing tool. It helps you determine variable and macro usage, analyze high-level control flows, and locates specific uses of arbitrary strings of characters.
- **Program Understanding Tool**, a workstation-based program analysis tool. It provides multiple and "variable-magnification" views of control flows within single programs or across entire application modules.
- **Interactive Debug Facility**, a powerful and sophisticated symbolic debugger for applications written in Assembler Language and other compiled languages. It simplifies and speeds the development of correct and reliable applications. (It is not intended for debugging privileged or supervisor-state code.)
- **Structured Programming Macros**, a complete set of macro instructions that implement the most widely used structured-programming constructs (IF, DO, CASE, SEARCH, SELECT). These macros simplify coding and help eliminate errors in writing branch instructions.
- **File Comparison Utility** (known as "Enhanced SuperC"), a versatile file searching and comparison tool. It can scan or compare single file or groups of files with an extensive set of selection and rejection criteria. Typical uses include comparing an original source file with a modified source file, or a pre-migration application output file with a post-migration output file. Newly added functions include "smart comparisons" of date fields to assist date "windowing".

Together, these tools provide a powerful set of capabilities to speed application development, diagnosis, and recovery.

This presentation provides an overview of the features and use of each of the six Toolkit components. They are based on tools that have been used widely and tested extensively inside IBM before being “packaged” in the High Level Assembler Toolkit.

Why Use the Assembler Toolkit?

- Preserve investments in applications, people, skills, and procedures
 - Enhance the productivity of people with specialized skills
- Improve product maintainability and simplify upgrades
 - Enhancement and maintenance average 60% of software costs
- Improve application understandability
 - Product understanding typically requires 30% of maintenance time
- Improve application error detection and correction
 - Normal testing typically covers only 60% of code paths
 - Even 100% coverage can't find the 75% of defects from...
 - missing logic paths that should have been there
 - combinations of paths that aren't tested by coverage tools
- The Toolkit components can provide savings in many areas

HLASM Toolkit Feature

© IBM Corporation 1995, 2003. All rights reserved.

2

Why Consider Using the Toolkit?

The six components of the High Level Assembler Toolkit Feature help you in managing all stages of application recovery, understanding, development, test, and maintenance. Among the reasons you may consider in using the Toolkit are:

1. Preserve investments in applications, people, skills, and procedures

Many organizations have substantial investments in applications or application components written in Assembler Language. Converting to other languages has many costs (many hidden, and many significant), so it is important to continue to maintain and enhance existing code. This also helps to preserve investments in personnel and their knowledge of the applications, as well as in the organization's established estimation, development, test, and maintenance procedures.

2. Improve application maintainability and understandability

Application maintenance is usually the largest cost element of an application, so several Toolkit components will be valuable in helping you with understanding and maintaining Assembler Language code.

3. Improve application error correction

Testing typically detects only a fraction of latent errors in applications before they are deployed; finding and fixing those problems is helped by the Toolkit.

The components of the High Level Assembler Toolkit Feature can help you save time, reduce costs, improve product quality, and increase customer satisfaction.

Hardware Requirements

The High Level Assembler Toolkit Feature requires the same hardware environments as IBM High Level Assembler for MVS & VM & VSE Version 1 Release 4. Requirements for 24-bit Virtual Storage are:

- Disassembler: 100K bytes
- IDF: 600K bytes
- XREF: depends on number and sizes of modules being scanned
- SuperC: depends on number and sizes of modules being scanned
- ...plus working storage (depending on the application)

The Program Understanding Tool (ASMPUT) component of the High Level Assembler Toolkit Feature requires a workstation capable of running OS/2, Windows 95, 98, 2000, or NT with a minimum of 16 MB memory (32 MB recommended) and 80 MB of available hard-drive space, plus a host-system connection or other means of transferring SYSADATA files to the workstation for analysis.

Software Requirements

The High Level Assembler Toolkit Feature operates in all MVS/ESA and VM/ESA environments where IBM High Level Assembler for MVS & VM & VSE Version 1 Release 4 (MVS & VM Edition) operates. On MVS, the Interactive Debug Facility's macro facilities require TSO/E V2 or later.

On z/OS and OS/390, the High Level Assembler Toolkit Feature is an optional element; it operates in all environments where the same level of the High Level Assembler base element operates.

The High Level Assembler Toolkit Feature operates in VSE/ESA Version 2 (or later) environments where IBM High Level Assembler for MVS & VM & VSE Version 1 Release 4 (VSE Edition) operates. On VSE, the Interactive Debug Facility requires VSE Version 2.2 or later.

The Toolkit Feature's components can be used independently of High Level Assembler. However, the most productive uses of many of the Toolkit Feature's components rely on SYSADATA files produced by High Level Assembler for MVS & VM & VSE.

Note: The SYSADATA files should not be created if the GOFF or XOBJECT option is in effect.

The Program Understanding Tool (ASMPUT) component of the High Level Assembler Toolkit Feature requires one of:

- OS/2* Version 4 (8H1425) with fixpack 8 or later
- Windows* 95
- Windows 98
- Windows 2000
- Windows NT Version 4.0 with Service Pack 3 or later, on Intel workstations only.
- Windows XP

A recommended host-connection software package is eNetwork Personal Communications Version 4.2.1 (8H8735), which supports OS/2 and Windows.

HLASM Toolkit Publications

GC26-8709 *Toolkit Feature Interactive Debug Facility User's Guide*

The reference document for all IDF facilities, commands, windows and messages.

GC26-8710 *Toolkit Feature User's Guide*

Reference and usage information for the Disassembler, the Cross-Reference Facility, the Program Understanding Tool, the File Comparison Utility, and the Structured Programming Macros

GC26-8711 *Toolkit Feature Installation and Customization Guide*

Information needed to install all Toolkit Feature components

GC26-8712 *Toolkit Feature Interactive Debug Facility Reference Summary*

Quick-reference summary, with syntax of all commands and a list of all options; for experienced ASMIDF users.

Publications

The four publications for the High Level Assembler Toolkit Feature are:

GC26-8709 *Toolkit Feature Interactive Debug Facility User's Guide*

The main reference document that describes all IDF facilities, commands, windows and messages.

GC26-8710 *Toolkit Feature User's Guide*

Reference and usage information for the Disassembler, the Cross-Reference Facility, the Program Understanding Tool, the Enhanced SuperC File Comparison Utility, and the Structured Programming Macros

GC26-8711 *Toolkit Feature Installation and Customization Guide*

Information needed to install all Toolkit Feature components

GC26-8712 *Toolkit Feature Interactive Debug Facility Reference Summary*

Quick-reference summary, with syntax for all commands and a list of all the options. This booklet is intended for experienced ASMIDF users.

For more information about ordering the High Level Assembler Toolkit Feature, refer to Software Announcement 295-498, dated December 12, 1995.

HLASM Toolkit Structured Programming Macros

- Macro sets can help eliminate test/branch instructions, simplify program structures:
 1. **If-Then-Else, If-Then** (IF/ELSEIF/ELSE/ENDIF)
 2. **Do, Do-While, Do-Until** (DO/ITERATE/DOEXIT/ASMLEAVE/ENDDO)
 - supports forward/backward indexing, FROM-TO-BY values, etc.
 3. **Search** (STRTSRCH/ORELSE/ENDLOOP/ENDSRCH/EXITIF)
 - supports flexible and powerful choices of loop controls and test conditions
 4. **Case** (CASENTRY/CASE/ENDCASE).
 - provides rapid switching via N-way branch to specified cases
 5. **Select** (SELECT/WHEN/OTHRWISE/ENDSEL) with two forms!
 - allows general choices among cases using sequential tests
- All macro sets may be (properly) nested in any order, to any level
- You can use the full instruction set (including the newest ops)

HLASM Toolkit Structured Programming Macros

The High Level Assembler Toolkit Feature Structured Programming Macros simplify the coding and understanding of complex control flows, and help to minimize the likelihood of introducing errors when coding test and branch instructions. The macros support the most widely used structured-programming control structures and eliminate the need to code most explicit branches.

The Toolkit Feature Structured Programming Macros can be used to create the following structures:

- IF/ELSEIF/ELSE/ENDIF
One-way or two-way branching, depending on simple or complex test conditions.
- DO/ITERATE/DOEXIT/ASMLEAVE/ENDDO and STRTSRCH/ORELSE/ENDLOOP/ENDSRCH
A rich and flexible set of looping structures with a variety of control and exit facilities.
- CASENTRY/CASE/ENDCASE
Fast N-way branching, based on an integer value in a register. Deciding which branch to take is made at the CASENTRY macro; a direct branch to the selected CASE is then done, followed by an exit to the ENDCASE macro.
There is no OTHRWISE facility within this macro set.
- SELECT/WHEN/OTHRWISE/ENDSEL
Sequential testing, based on sets of comparisons, expressible in two different forms. These macros create a series of tests that are evaluated in the order they are specified in the program. If a test is true, the WHEN section of code for that test will be executed, followed by an exit at the ENDSEL macro. If no test is satisfied, then the OTHRWISE section (if present) will be performed.

All the macro sets may be nested, and there are no internal limits to the depth of nesting. Tests made by the various ENDxxx macros ensure that each structure's nesting closure is at the correct level, and diagnostic messages (MNOTEs) are issued if they are not.

Structured Programming Macros: Why Use Them?

Many users report the following benefits:

- Improved code readability and understandability
- Faster application development
- Cleaner code
- Eliminating extraneous labels makes code easier to revise
- You can use the SP macros when and where appropriate
 - Introduce the macros incrementally
- APAR PQ69812 adds extensive generalizations and improvements
 - APAR PQ74641 changes LEAVE to ASMLEAVE (IMS problem) and allows easy renaming of any macro

Why Use the Structured Programming Macros?

Experience with Structured Programming Macros has shown many benefits, including

- Improved code readability and understandability

Since application understanding and maintenance has significant costs, the improvements provided by the macros can reduce those costs.
- Faster application development

The macros simplify logic and need fewer statements to write, which can therefore speed your development tasks.
- Cleaner and more readable code

The macros can help eliminate extraneous statements and statement labels that might clutter the logic of a program, so the code is easier to write and read.
- Incremental use

You can use as few or as many of the macros as you like, and when you like; they can be introduced incrementally into existing programs. Thus, you aren't forced to make major changes to your code to start taking advantage of the macros' benefits.

Recent enhancements include:

- The Structured Programming Macros can generate based or relative-immediate instructions, especially branch on condition instructions. If you want to use relative-immediate instructions, simply specify ASMMREL ON after the COPY ASMMSP statement.
- Major extensions have been made to the capabilities of key macro sets.
- Various helpful diagnostics have been added, including checks for correct nesting.

Structured Programming Macros: Usage

- All macros are contained in a single member, ASMMSP
 - Use COPY ASMMSP statement to initialize
 - Or specify PROFILE(ASMMSP) option
 - Packaging dictated by IBM naming rules/conventions
- User macros have meaningful mnemonics
 - Internal (non-user) macro names begin with ASMM
- Global variables now begin with &ASMA_ to prevent conflicts
- GC26-8710, *High Level Assembler Toolkit User's Guide*

Structured Programming Macros: Usage

To use the macros, you must code COPY ASMMSP within the source. This will define all the macros as inline macros. Once this has been done you can use all the macros described without any further limitations. Alternatively, the High Level Assembler PROFILE(ASMMSP) option could be used to automatically include the ASMMSP member into the source without requiring any source changes.

Due to IBM Corporate product-naming standards, all distributed part names must start with the product prefix. In the case of these macros, this resulted in the creation of the ASMMSP member which contains all the “high level” user macros such as IF, CASE, etc. All supplied members have a prefix of ASMM.

The “user” macros are grouped in the following five sets:

- IF/ELSEIF/ELSE/ENDIF
- DO/ITERATE/DOEXIT/ASMLEAVE/ENDDO
- STRTSRCH/EXITIF/ORELSE/ENDLOOP/ENDSRCH
- CASE/CASENTRY/ENDCASE
- SELECT/WHEN/OTHRWISE/ENDSEL

We will describe each of these sets in turn.

In many of the following examples, a test condition is shown as (a). A test condition may take any of these basic forms:

(numeric_condition_mask)	values from 1 to 14
(condition_mnemonic)	E, NE, H, NH, GT, LE,... etc.
(instruction,operand1,operand2,condition)	(LTR,0,0,Z)
(instruction,operand1,operand2,operand3,condition)	(CLM,R6,B'1100',XY,NE)
(compare_instruction,operand1,condition,operand2)	(CLI,CHAR,EQ,C'*')

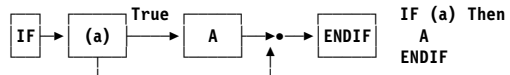
Multiple conditions may be combined using *logical connectors*: AND, OR, ANDIF, and ORIF, as in

(LTR,0,0,Z),OR,(CLI,CHAR,EQ,C'*')

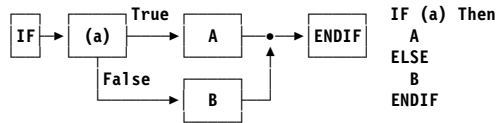
Very elaborate conditions can be constructed from the basic forms and connectors; see the *High Level Assembler Toolkit User's Guide* for details.

Structured Programming Macros: IF-THEN-ELSE

- Basic IF-ENDIF:



- Basic IF-ELSE-ENDIF:



- The word THEN is **not** syntactic; only a comment
 - Used only to improve readability, understandability

Structured Programming Macros: If-Then-Else

These “IF-THEN-ELSE” macros (IF/ELSEIF/ELSE/ENDIF) provide for one- or two-way branching depending on a condition. You may select execution of one of two blocks of code depending on a true-false condition.

The one-way IF-ENDIF branch is illustrated in Figure 17:

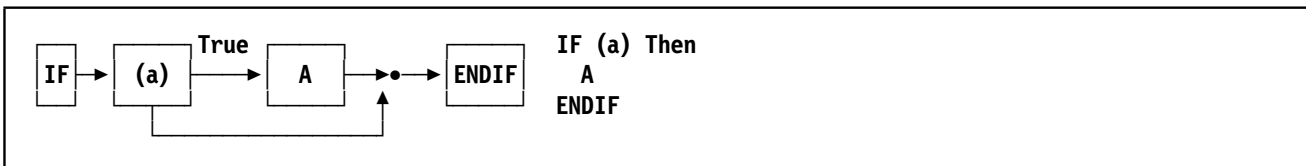


Figure 17. IF-THEN Control Structure

The two-way IF-ELSE-ENDIF branch is illustrated in Figure 18:

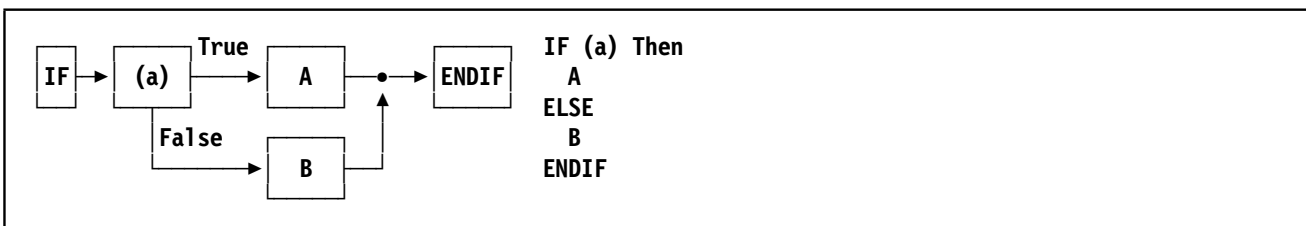


Figure 18. IF-THEN-ELSE Control Structure

Structured Programming Macros: Example 1

- Add absolute value of c(R4) to c(R5); don't change R4
- Unstructured:

```
LTR R4,R4          Set CC
BM LABEL1         Negative? Branch
AR R5,R4          Positive or zero - add to R5
B LABEL2          Skip the negative case
LABEL1 DS OH
SR R5,R4          Subtract negative value
LABEL2 DS OH
```

- Structured:

```
IF (LTR,R4,R4,NM) THEN Test R4 for non-negative
AR R5,R4               Positive or zero - add to R5
ELSE ,                 Otherwise,
SR R5,R4               Subtract negative value
ENDIF
```

- Can also use relative-immediate instructions:

```
IF (CHI,15,EQ,-3)      Compare with Halfword-Immediate
```

Structured Programming Macros: Example 1

This assembler program segment shows how to test a variable and then execute one of two paths depending on the value of the variable. The “problem” requires that we add the absolute value of the contents of R4 to R5, without disturbing R4.

This IF/ELSE/ENDIF structure is first coded using basic assembler language and then using the Toolkit macros. The unstructured assembler language segment could appear as follows:

```
LTR R4,R4          Set CC
BM LABEL1         Negative? Branch
AR R5,R4          Positive or zero - add to R5
B LABEL2          Skip the negative case
LABEL1 DS OH
SR R5,R4          Subtract negative value
```

The structured equivalent could be written as follows (remember that the THEN “keyword” is only a comment; it is not part of the syntax of the IF/ELSE macros):

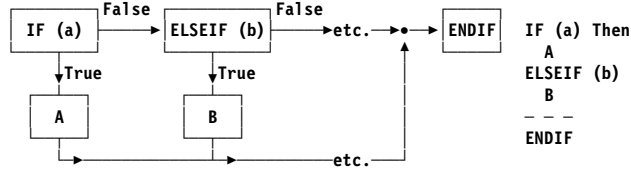
```
IF (LTR,R4,R4,NM) THEN Test R4 for non-negative
AR R5,R4               Positive or zero - add to R5
ELSE ,                 Otherwise,
SR R5,R4               Subtract negative value
ENDIF
```

and the results would be identical to the original (non-structured) statements:

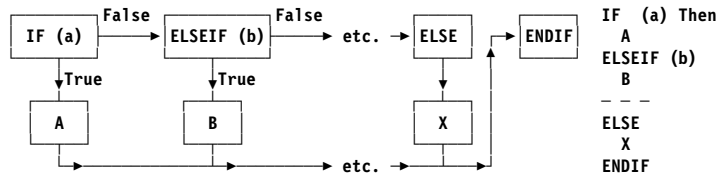
```
IF (LTR,R4,R4,NM) THEN Test R4 for non-negative
+ LTR R4,R4
+ BC 15-11,@LB1
AR R5,R4               Positive or zero - add to R5
ELSE ,
+ BC 15,@LB3
+@LB1 EQU *
SR R5,R4               Subtract negative value
ENDIF
+@LB3 EQU *
```

Structured Programming Macros: IF-THEN-ELSEIF-ELSE

- The ELSEIF macro simplifies deep nesting of IF-ELSE-ENDIF groups:



- Also used with an ELSE clause:



HLASM Toolkit Feature

© IBM Corporation 1995, 2003. All rights reserved.

26

Testing multiple conditions can be written as nested IF statements:

```

IF (a) Then
  A
ELSE
  IF (b) Then
    B
  ELSE
    - - -
    (more IF/ELSE/ENDIFs)
    - - -
  ENDIF
ENDIF

```

If the number of tests is large, the increased nesting levels can become awkward to manage. The ELSEIF macro can reduce the nesting of such structures to a single level, and can be used with or without an ELSE clause, as shown in Figure 19:

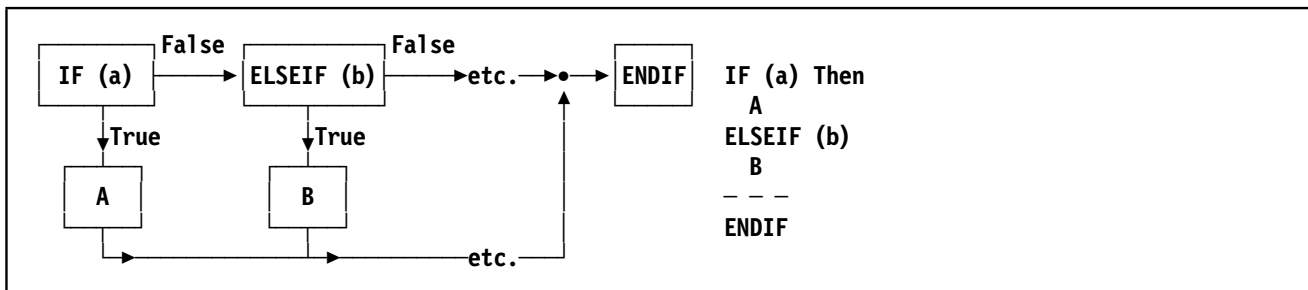


Figure 19. Simplified Structure with Multiple ELSEIF Statements

The same structure with an ELSE clause is shown in Figure 20 on page 39:

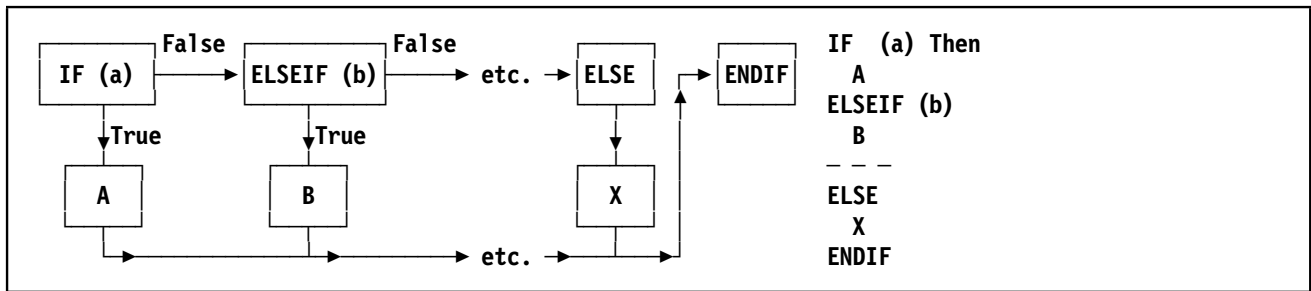


Figure 20. Simplified Structure with Multiple ELSEIF Statements and ELSE Clause

These four macros provide a complete set of conditional statement structures.

Structured Programming Macros: DO Set

- **DO, DO-WHILE, DO-UNTIL** predicates support mixtures of WHILE, UNTIL, forward/backward indexing, FROM-TO-BY values, etc.
 - DOEXIT macro uses IF-macro syntax to exit the containing DO
 - ASMLEAVE exits any number of containing labeled DOs
 - ITERATE requests immediate execution of the next loop iteration for any containing DO
- A *very* rich and flexible set of facilities
- Simplest form: infinite loop, exited with a DOEXIT macro

```

DO INF
  A
  DOEXIT (a)
  B
ENDDO

```

HLASM Toolkit Feature © IBM Corporation 1995, 2003. All rights reserved. 27

Structured Programming Macros: Do, Do-While, Do-Until

These macros provide for executing a block of code repeatedly until some limit is reached or some condition is satisfied (DO, DO-WHILE, DO-UNTIL macros). The conditions controlling the looping and the termination condition may be specified in a rich set of combinations:

- with FROM,TO,BY specifications, or with infinite looping
- by counting
- with forward or backward indexing
- with explicit specification of BXH or BXLE
- DO-WHILE and DO-UNTIL (or mixed with any other DO type)
- DOEXIT macro uses IF-macro syntax to exit the containing DO (one level)
- ASMLEAVE exits any number of containing DOs (one or more levels)
- ITERATE requests immediate execution of the next loop iteration

The simplest loop — the “infinite” loop, terminated by some internally-determined condition — is most conveniently exited using the DOEXIT macro, as shown in Figure 21 on page 40:

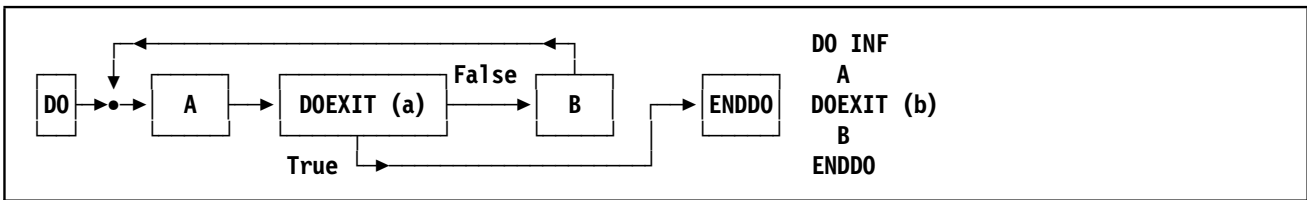
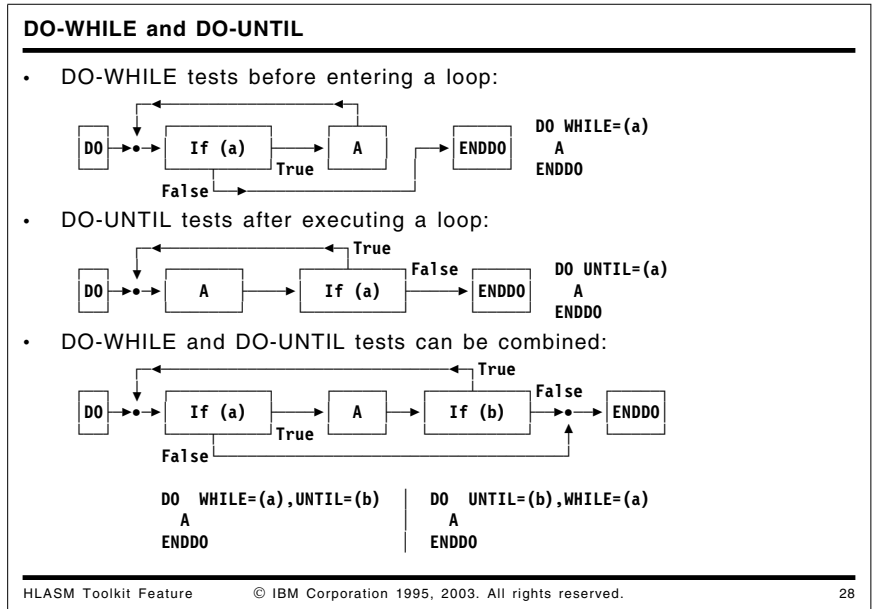


Figure 21. Simple DO Loop Structure with DOEXIT Statement



The DO-WHILE and DO-UNTIL control structures allow further conditional controls over loop execution. Their condition tests may be any operands valid on IF macros, except that the CC= operand is not allowed.

A DO-WHILE structure tests a condition before executing a loop, as illustrated in Figure 22 below:

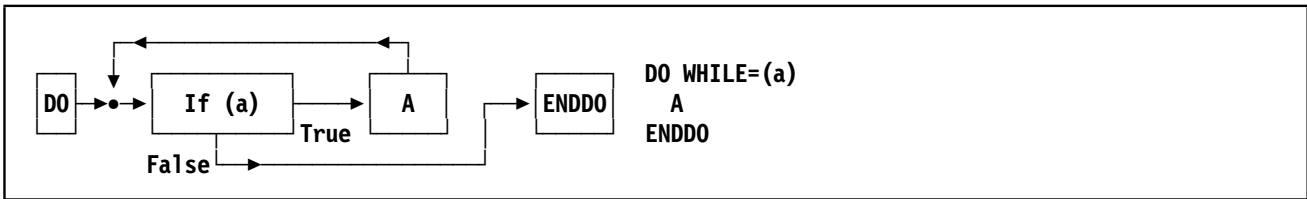


Figure 22. DO-WHILE Control Structure

A DO-UNTIL structure tests a condition after executing a loop, as illustrated in Figure 23 below:

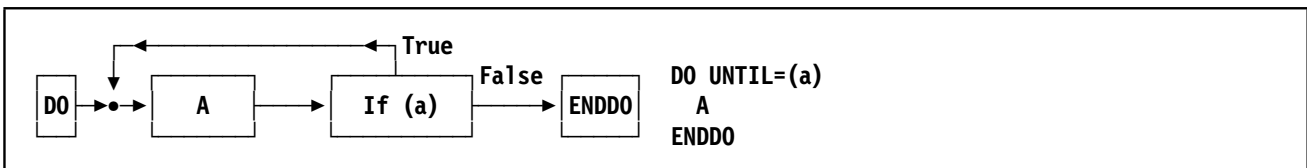


Figure 23. DO-UNTIL Control Structure

The DO-WHILE and DO-UNTIL structures can be combined, as illustrated in Figure 24 on page 41 below:

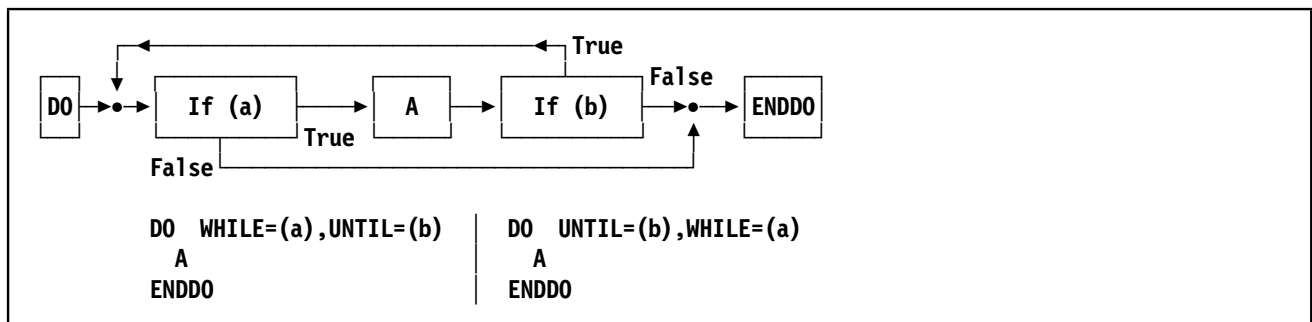


Figure 24. DO-WHILE/DO-UNTIL Control Structure

Structured Programming Macros: Example 2

- Search a string for first blank character, or end of string
- Unstructured:


```

L      R5,=A(Start-1)      Address start-1 of expression
Top_of_Loop DS 0H
C      R5,End              Test for end of expression
                        and exit if we've reached end
BNL   Leave_Loop
LA     R5,1(,R5)           Move along one byte
CLI   0(R5),C' '          Test for a blank
BNE   Top_of_Loop         not yet, repeat loop
Leave_Loop DS 0H

```
- Structured:


```

L      R5,=A(Start-1)      Address start-1 of expression
Scan  DO WHILE=(C,R5,LT,End),UNTIL=(CLI,0(R5),EQ,C' ')
      LA     R5,1(,R5)      Move along one byte
ENDDO

```

HLASM Toolkit Feature © IBM Corporation 1995, 2003. All rights reserved. 29

Structured Programming Macros: Example 2

This assembler program segment shows a simple loop that scans storage until either a blank is found or the end-of-string address is reached.

This DO/ENDDO structure is first coded using basic assembler language and then using the toolkit macros. The unstructured assembler language might appear as follows:

```

L      R5,=A(Start-1)      Address start-1 of expression
Top_of_Loop DS 0H
C      R5,End              Test for end of expression
                        and exit if we've reached end
BNL   Leave_Loop
LA     R5,1(,R5)           Move along one byte
CLI   0(R5),C' '          Test for a blank
BNE   Top_of_Loop         not yet, repeat loop
Leave_Loop DS 0H

```

The same example could be coded using the DO and ENDDO macros as follows:

```

L      R5,=A(Start-1)      Address start-1 of expression
Scan  DO WHILE=(C,R5,LT,End),UNTIL=(CLI,0(R5),EQ,C' ')
      LA     R5,1(,R5)      Move along one byte
ENDDO

```

Note that in both examples the required COPY ASMMSP statement is not shown.

Structured Programming Macros: Iterative-Do Macros

- Two styles: simple count, general indexing
- Count style does a set number of iterations

- Indexing form is extremely flexible

```
DO [BXH|BXLE,]FROM=(Rx,num),TO=(Ry+1,num),BY=(Ry,num)
A
ENDDO
```

- Counts up or down
- Automatic or user selection of BXH or BXLE loop closing
- Many variations supported

HLASM Toolkit Feature © IBM Corporation 1995, 2003. All rights reserved. 30

Structured Programming Macros: Iterative-Do Indexing Group

The iterative forms of DO statements can take five basic forms:

- the Count form loops a specified number of times, terminating the loop using a branch-on-count instruction
- four other forms use the FROM, TO, and BY keywords, as well as allowing explicit selection of B(R)XH and B(R)XLE instructions to terminate the loop.

The Count form is simplest to specify, as shown in Figure 25:

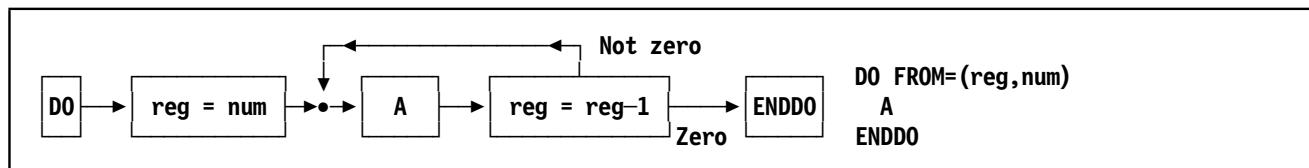


Figure 25. DO-FROM Counting Control Structure

The general form of an indexing DO macro is more complex, requiring specification of various combinations of parameters, as shown in Figure 26:

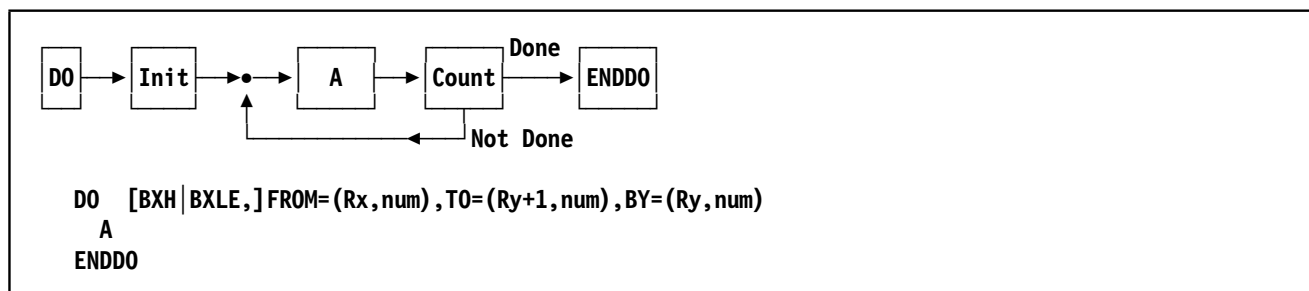


Figure 26. DO-FROM Counting Control Structure

The many allowed combinations are described in the *High Level Assembler Toolkit User's Guide*.

Structured Programming Macros: Exiting and Repeating Do Loops: ITERATE and ASMLEAVE

Often it is necessary to begin the next loop iteration before the remaining statements in the loop have been executed. This can be done with the ITERATE macro:

```

DO (various options)
  some code
  IF (a condition)      Then,
    ITERATE ,          causes next loop iteration
  ENDIF
  more code             skipped by ITERATE
ENDDO,                 ITERATE "branches" here

```

The ITERATE macro also allows you to exit its immediately enclosing DO structure in order to cause an iteration of an outer, enclosing DO structure. This is done by labeling the DO statement, and then using the appropriate label as an operand of ITERATE:

```

DoA DO (various options)      in outer loop
  some code in DoA
DoB DO (other options)       in inner loop
  some code in DoB           Then,
  IF (a condition)           causes next loop iteration in outer loop
    ITERATE DoA
  ENDIF
  more code in DoB           skipped by ITERATE DoA
ENDDO
  final code in DoA         skipped by ITERATE DoA
ENDDO ,                      ITERATE DoA "branches" here

```

Note that ITERATE DoB would be equivalent to simply specifying ITERATE with no operand.

Similarly, the ASMLEAVE macro allows you to exit its immediately enclosing DO structure, whether or not all the loop's iteration conditions have been satisfied.

```

DO (various options)
  some code
  IF (a condition)      Then,
    ASMLEAVE ,          causes loop termination
  ENDIF
  more code             skipped by ASMLEAVE
ENDDO
*                        ASMLEAVE "branches" here

```

ASMLEAVE, like ITERATE, also allows you to specify an operand naming a DO statement: the specified DO loop is exited, and control passes to the statement following its matching ENDDO.

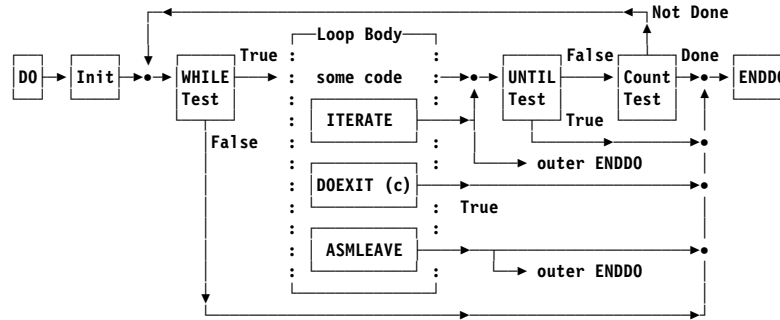
```

DoA DO (various options)      in outer loop
  some code in DoA
DoB DO (other options)       in inner loop
  some code in DoB           Then,
  IF (a condition)           Exits DoA loop (and DoB, also!)
    ASMLEAVE DoA
  ENDIF
  more code in DoB           skipped by ITERATE DoA
ENDDO
  final code in DoA         skipped by ITERATE DoA
ENDDO ,
*                        ASMLEAVE DoA "branches" here

```

Structured Programming Macros: General Form of DO Statement

- DO statement supports a rich combination of operands



- You can specify very detailed loop controls

Structured Programming Macros: General Form of Do Statements

The many DO options — WHILE, UNTIL, indexing, DOEXIT, ITERATE and ASMLEAVE, are sketched in Figure 27:

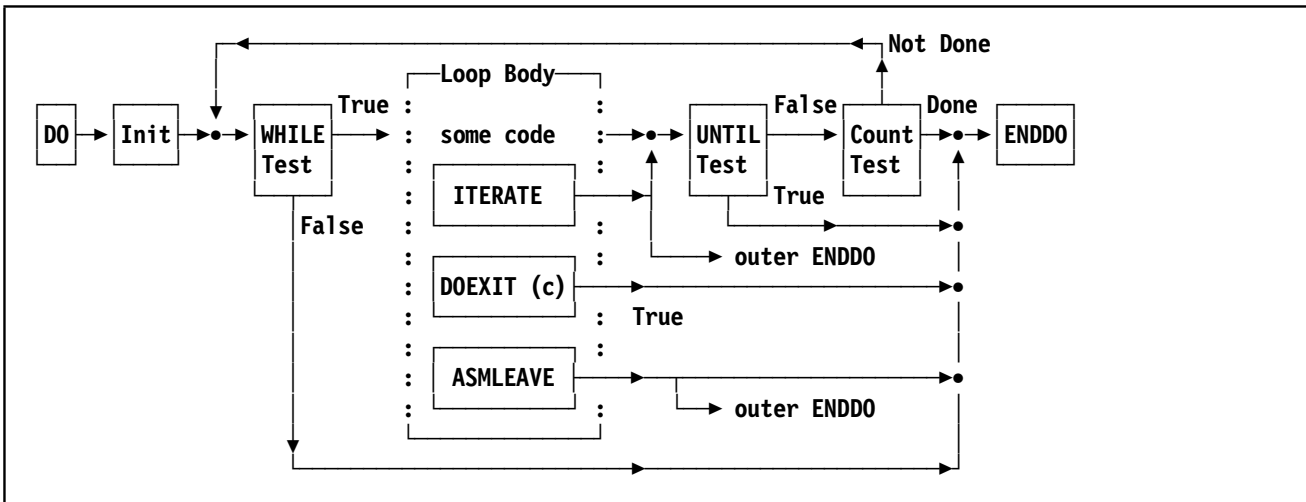


Figure 27. General Form of DO Control Structure

With them, you can write very flexible and complex looping structures.

Structured Programming Macros: SEARCH Set

- SEARCH set specifies a complex looping structure:

```

graph LR
    STRTSRCH --> A[A]
    A --> EXITIF["EXITIF (x)"]
    EXITIF -- true --> B[B]
    EXITIF -- ORELSE --> C[C]
    B --> TEST["test for end loop condition"]
    C --> TEST
    TEST -- Not Done --> A
    TEST -- ENDLOOP --> D[D]
    D --> ENDSRCH
  
```

- Statement format:

```

STRTSRCH (any DO-loop operands)
Process Code A
EXITIF (any IF-type operands)
Process Code B
ORELSE
Process Code C ] last one optional ] repeatable clauses
ENDLOOP
Process Code D
ENDSRCH
  
```

HLASM Toolkit Feature © IBM Corporation 1995, 2003. All rights reserved. 32

Structured Programming Macros: Search Set

The macros in the SEARCH set provide for executing a search loop with flexible controls over exit and iteration conditions.

```

STRTSRCH (any DO-loop operands)
Process Code A
EXITIF (any IF-type operands)
Process Code B
ORELSE
Process Code C ] last one optional ] repeatable clauses
ENDLOOP
Process Code D
ENDSRCH
  
```

Any number of EXITIF-ORELSE clauses may be specified; the last ORELSE clause preceding the ENDLOOP macro may be omitted.

The control structure supported by the Search Set is shown in Figure 28 below:

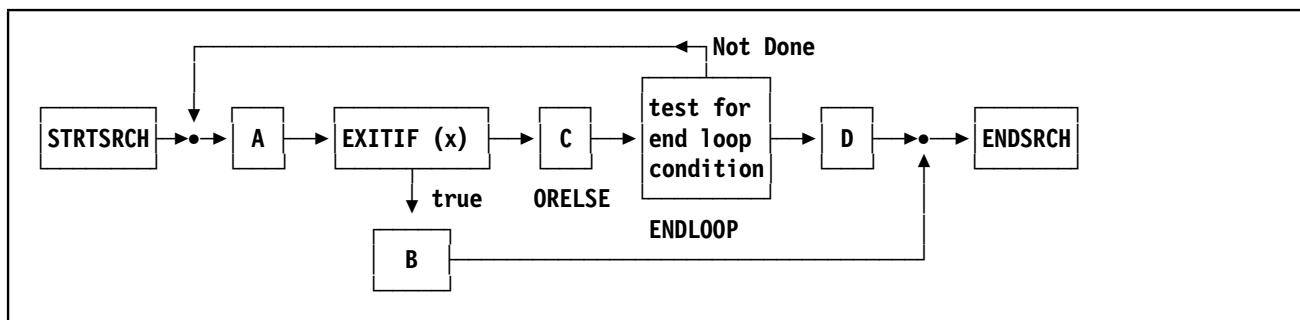


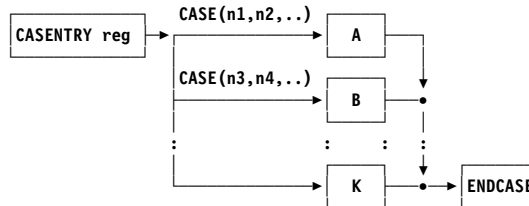
Figure 28. SEARCH Control Structures

Structured Programming Macros: CASE Set

- CASE macros provide rapid selection of blocks of code

```
CASEENTRY register[,POWER=p,VECTOR=B|BR]
CASE n1,n2,...
  Process Code A
CASE n3,n4,...
  Process Code B
-----
ENDCASE
```

- register operand contains an integer power of 2, **p**
- VECTOR operand selects table of branches, or adcons used by BR



HLASM Toolkit Feature

© IBM Corporation 1995, 2003. All rights reserved.

33

Structured Programming Macros: Case Set

These macros provide for executing a block of code selected from a set, based on an integer value contained in a general register. The integer value may also be a power of two, as specified by the optional POWER= keyword. The basic syntax is shown in Figure 29:

```
CASEENTRY register[,POWER=p,VECTOR=B|BR]
CASE n1,n2,...
  Process Code A
CASE n3,n4,...
  Process Code B
-----
ENDCASE
```

Figure 29. CASE Statement Syntax

The selected case is branched to directly, using one of two selection mechanisms depending on whether the branching should use a “vector” of addressable branch instructions (VECTOR=B) or a table of address constants (VECTOR=BR), one of which will be used by a BR instruction.

A contiguous set of CASE values need not be specified. Any “missing” cases between 1 and the largest case number simply generate a branch to the ENDCASE macro.

The CASE control structure is illustrated in Figure 30 below:

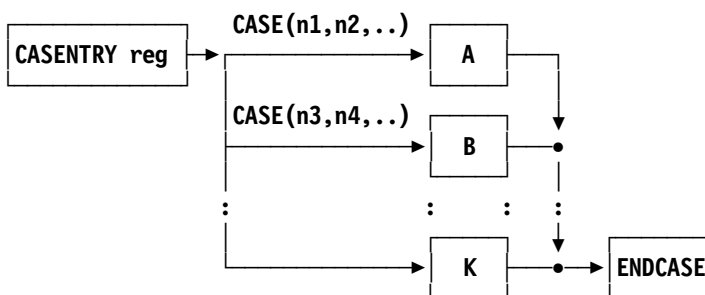


Figure 30. CASE Control Structures

A simple example of the CASE macros is the following:

```

CasEntry R1
  Case (1,2,3,5,7)
    MVI Flag,Prime
  Case (4,6,8)
    MVI Flag,NotPrime
EndCase
- - -
Flag    DC    X'0'
Prime   Equ   X'80'
NotPrime Equ  X'40'

```

Structured Programming Macros: SELECT Set

- SELECT group with single comparison:

SELECT (comparison)	Compare instruction & condition
WHEN (list-of-values-1)	Values for this comparison
<statements-1>	Statements for these cases
. . .	
WHEN (list-of-values-n)	Values for last comparison
<statements-n>	Statements for these cases
OTHRWISE ,	
<statements>	Executed if no matching WHEN
ENDSEL ,	End of SELECT group

- SELECT group with multiple comparisons/tests:

SELECT ,	No operands
WHEN (comparisons-1)	Comparisons and/or tests
<statements-1>	Statements for these cases
. . .	
WHEN (comparisons-n)	Comparisons and/or tests
<statements-n>	Statements for these cases
OTHRWISE ,	
<statements>	Executed if no matching WHEN
ENDSEL ,	End of SELECT group

HLASM Toolkit Feature © IBM Corporation 1995, 2003. All rights reserved. 34

Structured Programming Macros: Select Set

The SELECT group of macros — SELECT, WHEN, OTHRWISE, and ENDSEL — provide flexible techniques for choosing among alternatives by sequential testing. Two types are supported:

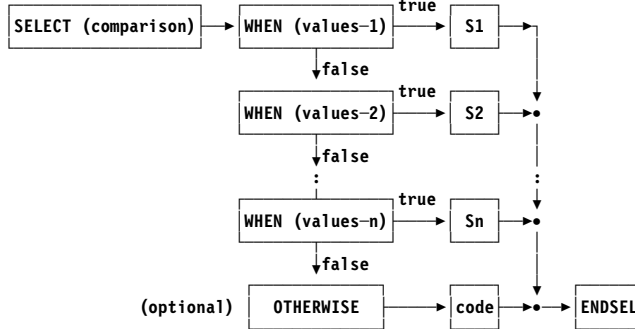
1. Single comparison: one comparison test is specified on the SELECT macro, and is used for testing the values specified in each WHEN clause.
2. Multiple comparison/test: no test is specified on the SELECT macro. Rather, each WHEN macro specifies the conditions that must be satisfied in order for its group of statements to be executed. This allows you to vary the sequence of tested conditions.

Note that the parentheses around SELECT comparison operands are usually optional, and no parentheses are required for a single WHEN operand.

(You may remember that testing for the most frequently occurring conditions first leads to increased efficiency.)

Structured Programming Macros: Single-Comparison SELECT

- Same comparison used for all WHEN clauses



- WHEN operand is a list of one or more items
- Easy way to test a series of identical data types

Structured Programming Macros: Single-Comparison SELECT ...

- Example: check for characters in arithmetic expressions

```
SELECT (CLI,Flag,eq)
  When (C'*',C'/',C'+',C'-')
  S1
  When (C'(',C')',C'=' )
  S2
  OTHRWISE
  code
ENDSEL
```

- Example: test small numbers in R1 for primes

```
SELECT C,R1,Eq
  WHEN =F'0'
  ErrorMsg 'Zero not a valid prime'
  WHEN (=F'1',=F'2',=F'3',=F'5',=F'7')
  MVI Flag,Prime
  WHEN (=F'4',=F'6',=F'8')
  MVI Flag,NotPrime
  OTHRWISE
  MVI Flag,Unknown
ENDSEL
```

The SELECT with a single comparison provides a simple way to make selections among operands of the same type. For example, checking for the occurrence of a character that might occur in an arithmetic expression is shown in Figure 31 on page 49:

```

SELECT  (CLI,Flag,eq)
  When  (C'*',C'/',C'+',C'-')
    S1
  When  (C'(',C')',C'=')
    S2
  OTHRWISE
    code
ENDSEL

```

Figure 31. Example of Single-Comparison SELECT

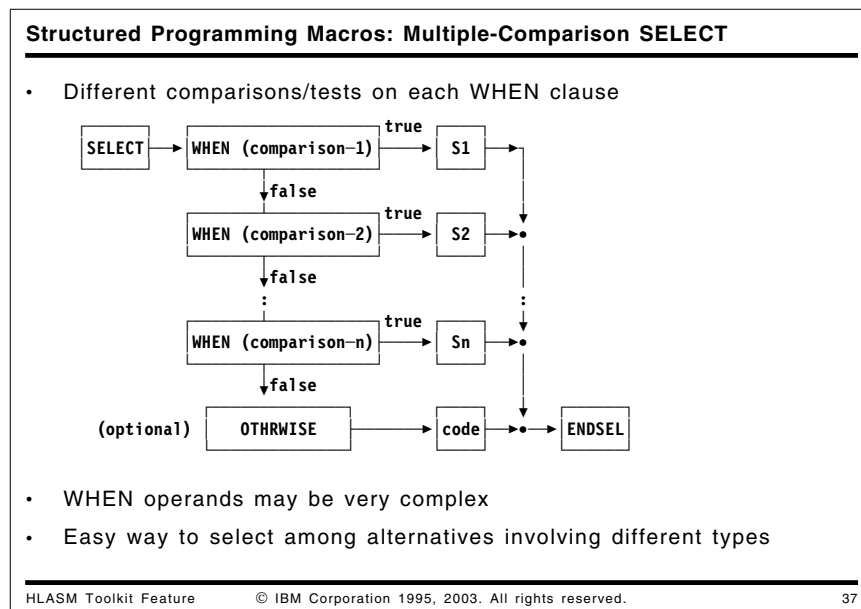
Another example of the SELECT macros using arithmetic comparisons:

```

Select  C,R1,Eq
  When  (=F'1',=F'2',=F'3',=F'5',=F'7')
    MVI  Flag,Prime
  When  (=F'4',=F'6',=F'8')
    MVI  Flag,NotPrime
  Othwise
    MVI  Flag,Unknown
EndSel
- - -
Flag    DC    X'0'
Prime   Equ   X'80'
NotPrime Equ  X'40'
Unknown Equ   X'01'

```

Figure 32. Example of Single-Comparison SELECT



The second style of SELECT group provides greater complexity in the test conditions used to choose the statements executed when the WHEN clause is true. The general format is shown in the following figure:

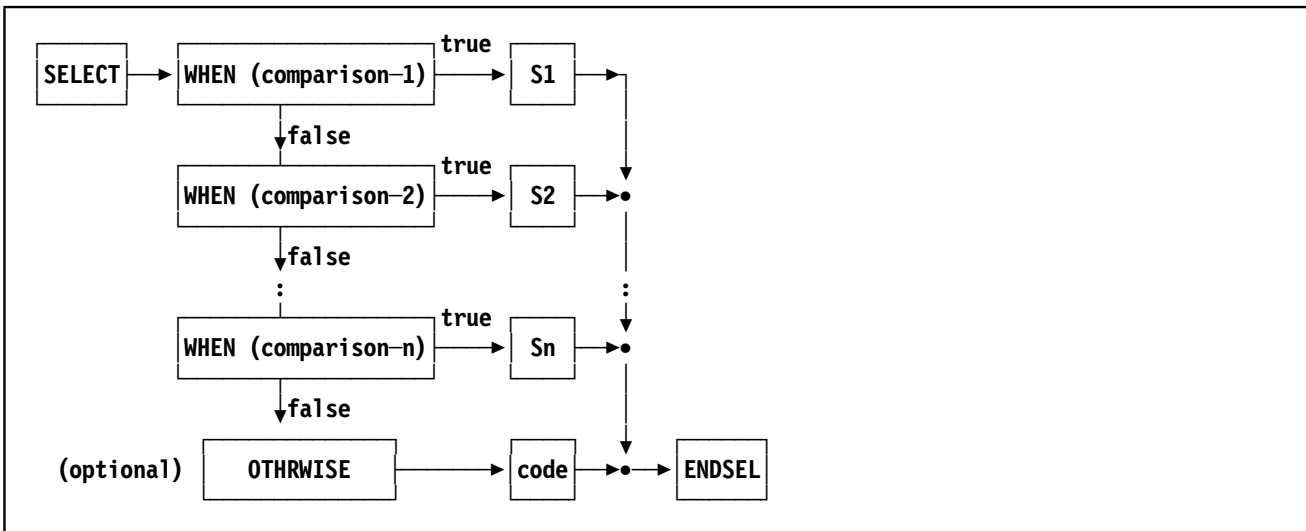


Figure 33. Multiple-Comparison SELECT Structure

Structured Programming Macros: Multiple-Comparison SELECT ...

- Example using mixed comparisons


```

SELECT
  When (CLI,Flag,eq,C'+'),Or,(CLI,Flag,eq,C' ')
  S1
  When (CLI,Flag,eq,C'-'),And,(LTR,R0,R0,M)
  S2
  ---
  OTHRWISE
  code
ENDSEL

```
- Example: test value in R1 for a small prime


```

ST R1,Temp
SELECT
  When (LTR,R1,R1,P),And,(C,R1,lt,=F'4')
  MVI Flag,Prime      R1 contains 1, 2, or 3
  When (TM,Temp,NZ,2)  Is it even?
  MVI Flag,NotPrime
  OTHRWISE
  MVI Flag,UnKnown
ENDSEL

```

HLASM Toolkit Feature © IBM Corporation 1995, 2003. All rights reserved. 38

An example using the second style of SELECT group is shown in Figure 34 below. It is similar in function to the example shown in Figure 31 on page 49, but uses more complex test conditions on each WHEN clause.

```

SELECT
  When (CLI,Flag,eq,C'+'),Or,(CLI,Flag,eq,C' ')
  S1
  When (CLI,Flag,eq,C'-'),And,(LTR,R0,R0,M)
  S2
  ---
  OTHRWISE
  code
ENDSEL

```

Figure 34. Example of Multiple-Comparison SELECT

As a final example of the SELECT group, the code fragment in Figure 32 on page 49 is revised to use multiple-comparison WHEN clauses:

```

ST    R1,Temp
SELECT
  When  (LTR,R1,R1,P),And,(C,R1,lt,=F'4')
        MVI  Flag,Prime      R1 contains 1, 2, or 3
  When  (TM,Temp,NZ,2)    Is it even?
        MVI  Flag,NotPrime
  OTHRWISE
        MVI  Flag,UnKnown
ENDSEL

```

The generated instructions from this code fragment is:

```

ST    R1,Temp
SELECT
  When  (LTR,R1,R1,P),And,(C,R1,lt,=F'4')
+      LTR      R1,R1
+      BRC      15-2,@LB2
+      C        R1,=F'4'
+      BRC      15-4,@LB2
        MVI  Flag,Prime      R1 contains 1, 2, or 3
  When  (TM,Temp+3,1,NZ)  Is it even?
+      BRC      15,@LB1      SKIP TO END
+@LB2      DC  0H
+      TM        Temp+3,1
+      BRC      15-7,@LB4
        MVI  Flag,NotPrime
  OTHRWISE
+      BRC      15,@LB1      SKIP TO END
+@LB4      DC  0H
        MVI  Flag,UnKnown
  ENDSEL
+@LB1      DC  0H

```

The SELECT macros provide for executing a block of code selected from a set of blocks, based on a choice of ordered comparisons.

While they appear to be structurally similar to the CASE set, their behavior is quite different. Each WHEN clause is tested in the order specified until a “true” condition is found, when the corresponding block of statements will be executed; the optional OTHRWISE block is executed if no test in any WHEN clause is true.

Structured Programming Macros: Detailed Example

- An elaborate example is provided in the text
 - Illustrates all of the macros, and all their options
 - Nested in various combinations

Source See Appendix A, "Sample structured macro program"

Listing See Appendix B, "Listing of sample program"

Structured Programming Macros: Extended Example

An extensive sample program is provided in Appendix A, "Sample structured macro program" on page 64. It shows the use of more complicated structures and the nesting of macros. (Note that no macros from the SELECT set are illustrated.)

The assembly listing is provided in Appendix B, "Listing of sample program" on page 67. The listing was created by the following (CMS) commands:

1. Access the High Level Assembler Toolkit disk
2. GLOBAL MACLIB ASMSMAC
3. ASMAHL SAMPLE (PROFILE(ASMMSP),NOESD,NORLD,NOXREF,NOMXREF,NOUSING

The expansion of the macros is shown in the listing; if this expansion is not desired then you may use the PC(NOGEN) option to suppress the generated lines.

Structured Programming Macros: Notes

- To generate relative branches, code ASMMREL ON (OFF for based)
 - Base register not required for generated code!
- Be **very** careful about continuations! (Run with FLAG(CONT) option)
- Boolean expressions partially optimized
 - Evaluated only as far as necessary to determine result
 - Can sometimes be simplified: NOT (A AND B) = ((NOT A) OR (NOT B))
- Limitation to at most 50 operands on any one macro
 - Parentheses in operands are optional, but helpful
- Some macro operand “keys” not safely usable as program symbols:
P, M, O, Z, H, L, E, NP, NM, NO, NZ, NH, NL, NE,
GT, LE, EQ, LT, GE, AND, OR, ANDIF, ORIF
- IF, DOEXIT, EXITIF, WHEN macros allow CC= as only operand

HLASM Toolkit Feature

© IBM Corporation 1995, 2003. All rights reserved.

40

Structured Programming Macros: Notes

The Structured Programming Macros can generate either based or relative branch instructions. To obtain the latter, just code the ASMMREL macro. You can switch back and forth between generating based and relative branches by coding the ON and OFF operands of ASMMREL.

Some minor points are worth remembering:

- Be very careful to place any continued operands in the correct column. The normal assembler rules apply (along with any changes that the ICTL statement may have introduced). The assembly-time option FLAG(CONT) can help determine where the rules have not been followed.
- Not only are the instructions generated by the macros nearly optimal, the macros do not need to evaluate all the terms in a Boolean expression before branching. In the following statement:

```
IF (LTR,R5,R5,P),AND,(LTR,R6,R7,P)
```

the second load and test (LTR) instruction will *not* be executed if the first LTR sets a negative or zero condition code, as the macros “know” that the expression must return false after only the first part has been evaluated.

A small reminder about Boolean logic: you can sometime simplify the operands of a test by rewriting expressions:

```
NOT (A AND B) is equivalent to ((NOT A) OR (NOT B))
```

- Most of the original limitations of these macros have been removed. (They were caused by previous assemblers having fixed array sizes; in HLASM, arrays are dynamic in nature and will grow as required.) One limitation remains: Boolean expressions are limited to fifty (50) operands. This count includes any operators such as AND, OR, etc.
- The use of parentheses in Boolean expressions is optional, but may assist with the understanding of the logic.
- Some “keys” required for correct operand parsing should not be used as ordinary program symbols: P, M, O, Z, H, L, E, NP, NM, NO, NZ, NH, NL, NE, GT, LE, EQ, LT, GE, AND, OR, ANDIF, ORIF.
- Four macros — IF, DOEXIT, EXITIF, and WHEN — support a CC= operand. If used, no other operands may be present.