1 Introduction

1.1 Brief product description

ASSEMBH is a language processor for the assembly and macro languages.

ASSEMBH processes one source program at a time. The text of a source program is a series of instruction statements and remarks which may represent one or more assembly units. As a rule, an assembly unit in an assembler source program begins with a START or CSECT instruction and ends with an END instruction. In structured programming (not supported by the ASSEMBH-BC basic package), these instructions are generated by means of corresponding predefined macros. ASSEMBH creates an object module for each assembly unit.

The following are examples of instructions in a source program:

– Assembler instructions;
  These cause the assembler to execute specific operations during assembly.

– Machine instructions;
  These cause the central processing unit to execute specific operations during the program run.

– Macro instructions (macro calls);
  These cause the assembler to read in pre-coded source texts during assembly. They are modified according to the information specified in the macro instruction, and the resulting instruction statements are inserted in the source program.

– Macro instruction statements;
  With these, the text and number of generated instructions may be varied on the basis of conditions evaluated during assembly.

The assembler instructions, macro calls, macro instruction statements, and predefined macro calls for structured programming are described below. A detailed description of the assembler instructions can be found in the manual "Assembler Instructions (BS2000), Reference Manual" [3].

1.2 Target group

A basic knowledge of the assembly language is required in order to use this manual effectively. It is designed not only as a reference manual, but also as a tutorial with which new functions can be learned and existing knowledge extended.

1.3 Summary of contents

The manual is divided into three sections:

- Chapters 2 to 4 describe the structure of the assembly language and the assembler instructions.
- Chapters 5 to 8 describe the structure, elements, and instructions of the macro language.
- Chapters 9 and 10 explain the structure of an assembly language program according to the rules of structured programming and explain the predefined macros for structured programming.

Operation of the ASSEMBH in BS2000 is described in the "ASSEMBH (BS2000) User Guide" [1]. The debugging system AID (Advanced Interactive Debugger) is described in the manual "AID (BS2000), Debugging of ASSEMBH Programs" [2].

References to other publications are given in abbreviated form in the text. The full title of each publication referred to can be found in the "References" section at the back of the manual. The relevant reference guide is also listed there.

Notes on ordering publications are provided at the end of the References section.

1.4 Changes since the last version of the manual

Isolated corrections made throughout the manual are not separately listed here.
The significant additions and modifications are as follows:

The functionality of the PUNCH and REPRO instsructions is not supported for modules in LLM format (section 4.2).

The following changes have been made to the @ macro descriptions in Chapter 10:
- operand description in @CONEN macro improved
- DROP operand description in END macro extended
- description of reg1-reg3 operands in @ENTR macro dropped
- RC operand description in @EXIT macro extended
- aadditions to the @ININ macro description
- event name 'INTR' added to the @STXEN macro description

The list in Appendix 11.2: 'Format of the machine instructions' has been extended to include the ESA instructions.

The table in Appendix 11.6: "Differences between ASSEMBH and ASSEMB" has been updated.

Any functional changes and additions to the current product version can be found in the chapter "Manual supplements".

1.5 Notational conventions

The following notational conventions are used in the instruction formats:

CONSTANTS	Uppercase letters denote metaconstants which must be entered in this form.
name	Lowercase letters denote metavariables for which the value relevant to the context must be entered.
YES	Underlined values denote default values that are entered by the assembler or the operating system.
⎡YES⎤ ⎨ ⎬ ⎣NO ⎦	Braces enclose alternatives. One of the specified values must be selected. The alternatives are listed one below the other. If one of the alternatives is a default value and the default value is desired, no specification is required.
[ ]	Square brackets enclose optional specifications which may be omitted.
( )	Parentheses are metaconstants and must be included in the entry.
...	Ellipses are used to indicate that the preceding unit can be repeated more than once.
[,...]	A comma followed by ellipses means that the preceding unit may be repeated more than once, but must be separated by a comma each time. The square brackets indicate that the specification is optional.
{}[,...]	In this instance, the single-line braces enclose the syntactical unit which may be repeated.
␣	Alternative representation of a space character. Used where a space character is syntactically necessary.

2 Assembly language structure

The text of an assembler source program consists of a series of instruction statements and remarks.
Here, instruction statements may be assembler instructions, machine instructions or macro instructions. Remarks are used for program documentation and have no effect on the assembled program.
In addition to these assembly language elements (described later), various macro language elements are permitted in the assembler source program. This facility is described in chapter 8.

2.1 Character set

The following characters may be used when entering instruction statements:

Letters	Digits	Special characters
A through Z	0 through 9	+ - * = , . () /
a through z if the option LOW-CASE-CONVERSION=YES has been used (see [1])		’ (single quote)
		& (ampersamd)
_ (underscore)		␣ (space character)
$ where names with $ as the first character are reserved for operating system applications (see [9])		
#		
@		

Examples of character set usage

Character	Usage	Example
Alphanumeric	in symbols	ADR100, AB_CD
Digits	as decimal self-defining terms	4096 8192
Special chars.	as operators:	
+	addition	AREA1+AREA2
-	subtraction	OUT-20
*	multiplication	3*ALPHA
/	division	NINE/3
+ or -	unary	+10, -4
	as delimiters:	
␣ Space	between entries	ADR1␣LR␣R5,R6
, Comma	between operands	OPND1,OPND2
’ Single quote	- to enclose data constants	C’CONSTANT’
	- in an attribute reference	L’AREA
( ) Parentheses	to enclose expressions and address constants	MVC AREA(12), (A+B*(C-D))
	as indicators of:	
. Period	- sequence symbols,	.LOOP
	- remark which does not go into the assembler listing	.*THIS IS A COMMENT
	- decimal point,	DC F’3.7C2’
	- concatenations	&PARAM.SAVE
* Asterisk	- location counter reference	*+100
	- remark that is entered in the assembler listing	*THIS IS A COMMENT
=	- literals,	MVC FELD,=C’SIEMENS’
	- keyword operands	MACALL &PAR=’ABC’
&	variable symbols	&PARAM

Uppercase and lowercase letters in source program text

With ASSEMBH, uppercase and lowercase letters can be used in instruction statements and remarks. provided the LOW-CASE-CONVERSION=YES option has been used (see ASSEMBH, User Guide [1]). Please note:

Lowercase letters are converted into uppercase and processed accordingly
– in symbols in the name and operand entries,
– in the operation entry and
– in assembler keywords (e.g. READ, PRVLGD,...).

Lowercase letters are processed unaltered
– in the remarks entry,
– in remarks lines,
– in C-type constants,
– in character self-defining terms, and
– in macro-language character values.

The original line is shown in the assembler listing in uppercase and lowercase letters. Source program lines generated by means of a macro instruction are shown in uppercase only.

Reference lists are shown in uppercase only.

2.2 Assembler instruction statements and remarks

Instruction statements

Instruction statements may consist of five entries:

– the name entry,
– the operation entry,
– the operand entry,
– the remarks entry and
– the continuation character

These entries must conform to the above order and be separated from one another by at least one space (except for the continuation character). Any number of continuation lines are permitted for an instruction statement (see section 2.2.5, "Continuation character").

The defaults for the begin, end, and continue columns are 1, 71, and 16, respectively. These defaults can be altered via a compiler option (see "ASSEMBH User Guide" [1]) or via the ICTL instruction (see section 4.2, ICTL instruction).

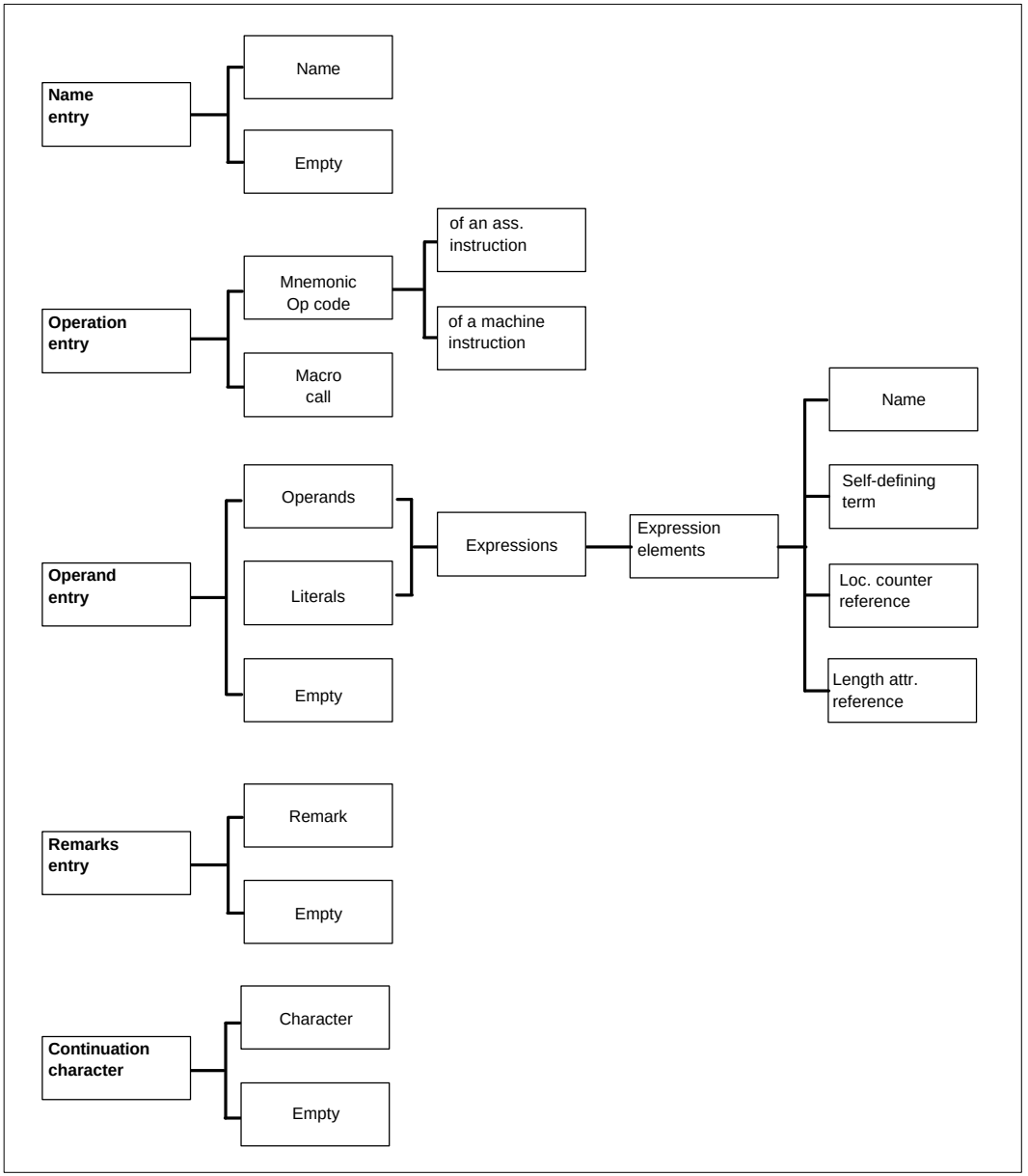


Fig. 2-1 Structure of Assembly language instructions

Remarks

Remarks are used for program documentation. They have no effect on the assembled program.

There are two types of remarks line:

* With an asterisk in the begin column;
These remarks lines are read by the assembler and printed out in the assembler listing.

.* With a period in the begin column, followed by an asterisk;
These remarks lines are not read by the assembler and therefore do not appear in the assembler listing.

There are no continuation lines for remarks. Any continuation character which has been entered is ignored. Longer remarks must be entered using several remarks lines.

Examples

```
Name        | Operation | Operand                                  | Continuation char.
------------------------------------------------------------------------------------
*  YOU NEED TWO LINES                                              X (is ignored)
*  FOR THIS COMMENT
.*  THIS COMMENT IS NOT PRINTED IN THE LISTING
```

2.3 Name entry

In the name entry is a name - with a maximum of 64 letters and digits - which identifies an instruction statement.

External names in modules in OM (object module) format, which are processed by the TSOSLNK linkage editor (see section 4.2, COM, CSECT, DXD, ENTRY, EXTRN, WXTRN, and XDSEC instructions), are limited to eight characters. Longer external names are truncated to eight characters for further processing and provided with a message.

External names in modules generated using the COMPILER-ACT(,MODULE-FORMAT=LLM) option (see ASSEMBH, User Guide [1]) are expanded to a length of 32 characters and are processed by the BINDER linkage editor.

The name entry may be optional. If it exists, it must start in the begin column. If the begin column is blank, the assembler assumes that no name exists and interprets subsequent characters as an operation code.

The assembler allocates a location counter to each name in the name entry. The same name in the name entry may be defined only once in an assembly unit. Control section names may occur more than once, as control sections may be interrupted and continued at another point in the program by repeating the section name (see also section 4.2, CSECT, DSECT, AMODE and RMODE instructions).

Location counters or names are generally relative; i.e. can have different values at the time of execution and the time of assembly. However, if an EQU instruction is used to assign an absolute value to a name, the value of the name does not change at execution time. The name is then referred to as a name with an absolute value.

Rules

- The first character of a name must be a letter.
- The value of a name must lie between -2^{31} and $2^{31}-1$.
- Underscores are permitted within the name (e.g. A_B), except in names processed by the linkage editor.
- No space characters are permitted within the name.
- The special characters & (ampersand) and . (period) have a special function (see chapter 5, Macro Language Structure).

Examples of valid names

```
LOOP                Field
A23456              @B4
X4F2                $A1
LOOP2               #56
LOOP_2              N
THIS_EXTREMELY_LONG_NAME_IS_REALLY_NOT_TOO_LONG_FOR_ASSEMBH
```

Examples of invalid specifications in the name entry

```
256B                (first character not a letter)
BCD*34              (contains the special character *)
IN␣PUT              (contains a space)
BUT_NOW_THIS_EXTREMELY_GREAT_LONG_NAME_IS_TOO_LONG_EVEN_FOR_ASSEMBH
```

Definition of names

A name is defined once it occurs in the name entry of an assembler instruction or machine instruction, or as an operand of an EXTRN or WXTRN instruction.

The name definition contains the implicit assignment of a length attribute. The length attribute of a name is the length of the designated memory area in bytes. Thus, a name which designates an RX instruction, for example, has the length attribute 4.

Exception

If a name has been equated with the location counter or with a self-defining term, the length attribute of the name is 1 (see section 4.2, EQU instruction, and 5.5.8, "Attribute references").

The length attribute of a name is not affected by a duplication factor.

2.4 Operation entry

The operation entry may include:

– the mnemonic operation code of an assembler or machine instruction or

– the name of a system or user macro (macro call).

An operation entry must be specified. At least one space character must separate it from the name entry. If there is no name entry, it must begin at least one position to the right of the begin column.

A valid operation entry consists of a maximum of five characters for assembler instructions and machine instructions. If the mnemonic operation code has been redefined via an OPSYN instruction, a maximum of 64 characters is allowed. A maximum of 64 characters is permitted for macro names as well.

There must be no space characters within the operation entry.

The OPSYN instruction (see section 4.2, "Description of instructions") can be used to alter the standard setting of the operation code and to create new mnemonic operation codes.

2.5 Operand entry

The operand entry contains the operands which describe or refer to the memory areas, masks, lengths, or data types to be processed.

If an operand entry is specified, it must be separated from the operation entry by at least one space character.

The operand entry may consist of one or more operands, which in turn may contain one or more expressions. Expressions comprise elements and operators.

Operands must be separated by commas. There must be no space characters between the operands and the separating commas.

An operand must not contain space characters.

Exception

Space characters in character constants that are used as literals, direct operands, or in a constant definition.

2.5.1 Expressions

Expressions are the basic operand components of instruction statements. They are used in order to calculate a value. Expressions are in turn made up of elements and operators. They may occur in the form of simple and arithmetic expressions in assembly language.

2.5.1.1 Simple expressions

Simple expressions consist of one element only. The value of the expression is equal to the value of the element.

The value of a simple expression must lie between -2^{31} and $2^{31}-1$

2.5.1.2 Arithmetic expressions

Arithmetic expressions are composed of elements and arithmetic operators.

The following arithmetic operators are allowed:

+ addition
– subtraction
* multiplication
/ division
+ unary plus
– unary minus

The sequence in which an arithmetic expression is evaluated can be controlled by the use of parentheses, which may also be nested.

Rules

– An arithmetic expression must not begin with an operator, except for unary plus and unary minus.

  Examples

  correct: `-7*(A+B)` incorrect: `*A+15`

– No two elements in an arithmetic expression must follow one another in succession.

Examples

correct: `FIELD1*(A+B)` incorrect: `FIELD1(A+B)`
`15*L'FIELD` `15L'FIELD`

– Unary plus and unary minus may directly follow all other operators.
– An arithmetic expression must not contain a literal.
– Values of arithmetic expressions must lie between -2^{31} and $2^{31}-1$.

Examples

Simple expressions	Arithmetic expressions
FIELD C'FIELD' X'4040' L'FIELD *	*+32 FIELD-35 FIELD*10 FIELD/2 FIELD1+FIELD2 (OUT-(IN*L'FIELD+1)+FIELD)

Evaluation of arithmetic expressions

Arithmetic expressions are evaluated according to the following rules:

1. Each element is assigned its own value.
2. Arithmetic operations are executed from left to right. Multiplication and division are done before addition and subtraction.
3. In expressions containing parentheses, the values enclosed in them are evaluated first. In the case of nested parentheses, the inner parenthesized expression is evaluated first.
4. Each arithmetic expression is calculated to a precision of 32 bits.
5. Remainders obtained from divisions are ignored. The result of every division is therefore an integer.

   Example

   ```
   1/(2*10)       gives         0
   -11/2          gives         -5
   ```

6. Division by zero is allowed and returns a result of zero.

2.5.3.1 Absolute and relocatable expressions

An expression is termed **absolute** if its value at program execution time is the same as its value at assembly time.

An expression is termed **relocatable** if its value at program execution time may differ from its value at assembly time.

The elements from which an expression is formed determine whether it is relocatable or absolute.

Absolute expressions

An absolute expression is reduced to an absolute value.

An absolute expression may be:

- a simple absolute expression,

  Example `L'FIELD`

- an arithmetic expression with only absolute elements,

  Example `L'FIELD+15`

- an arithmetic expression with a pair of relocatable elements with opposite signs,

  Example `*-FIELD`

- an arithmetic expression with relocatable and absolute elements.

  Example `(*-FIELD)*L'FIELD`

All arithmetic operations are permitted with absolute elements.

If an absolute arithmetic expression is to contain relocatable elements, the following conditions apply:

- The arithmetic expression must contain an even number of relocatable elements.
- The relocatable elements must be paired. In other words, the two relocatable elements belonging to a pair must be defined in the same control section and have opposite signs. They must not follow each other directly.
- No multiplication or division is permitted with a relocatable element.

The **occurrence of paired relocatable elements** in an absolute expression neutralizes the effects of program relocation at execution time. The value that is made up of the paired relocatable elements therefore remains constant, regardless of any program relocation.

Example

AA=AX+RY-RZ

		Value at assembly time	Value at execution time, e.g.
AX	absolute element	50	50
RY	relocatable element	10	110
RZ	relocatable element	25	125
AA	absolute expression	35	35

Relocatable expressions

The value of a relocatable expression is modified by n if the program in which it occurs has a load address incremented by n at execution time.

A relocatable expression may be:

– a simple relocatable expression,

  Example `*`

– an arithmetic expression with only relocatable elements,

  Example `*-FIELD1-TERM1`

– an arithmetic expression with relocatable and absolute elements.

  Example `*-FIELD1-TERM1+L'FIELD`

If a relocatable arithmetic expression is to contain only relocatable or relocatable and absolute elements, the following conditions apply:

– First option: the arithmetic expression contains an uneven number of relocatable elements, and these elements, except for one, are paired (see above).

  Example `*-FIELD1-TERM1`

  * and FIELD1 are to appear paired, i.e. they are defined in the same control section and have opposite signs.

– Second option: several relocatable elements are defined in the same control section and have no opposite signs. The result of their evaluation must then also be in the same control section.
  In addition to these single relocatable elements, the arithmetic expression may also contain paired relocatable elements.

  Example `TERM1+TERM2+TERM3+*-FIELD1`

  TERM1, TERM2, and TERM3 are defined in the same control section but do not occur in pairs. The result of their evaluation must then also be in the same control section.

– No multiplication or division is permitted with any of the relocatable elements.

A relocatable expression is reduced to a relocatable value. This is calculated from the value of the unpaired relocatable elements, modified by the value of the absolute and paired relocatable elements.

Example

RA=RU-RV+RU-10

		Value at assembly time	Value at execution time, e.g.
RU	relocatable element	10	110
RV	relocatable element	5	105
10	absolute element	10	10
RU-RV	paired reloc. elements	5	5
RA	relocatable expression	5	105

2.5.2 Elements of expressions

An element of an expression is the smallest assembly language unit which represents a value.

Each element corresponds to a value. This value is either assigned by the assembler (name, length attribute, location counter) or is directly contained in the element (self-defining term).

Each element may represent a simple expression only, or be combined with others into an arithmetic expression.

An element is termed **absolute** if its value is independent of the program load address, i.e. if its value at execution time is the same as its value at assembly time.

An element is termed **relocatable** if it is relative to the program start, i.e. if it may have a different value at execution time than at assembly time.

If two relocatable elements are combined into one expression, they may appear **paired**. This means that these two elements must be defined in the same control section, and have opposite signs.

The following elements of expressions occur in assembly language:

- name (absolute or relocatable)
- self-defining terms (absolute)
- location counter references (relocatable)
- length attribute references (absolute)

The elements of expressions and the rules for their usage are detailed below.

2.5.2.1 Names

The details given in section 2.2.1 for names in the name entry are also applicable to names in the operand entry.

2.5.2.2 Self-defining terms

Self-defining terms are used for direct representation of values. They are used to specify data, masks, registers, and address increments. Self-defining terms may be one word long at most.

Self-defining terms are absolute elements, as their value at execution time is the same as their value at assembly time.

Self-defining terms include:

- decimal,
- hexadecimal,
- binary, and
- character values

Examples

Self-defining term	Decimal value	Binary value	
241	241	1111 0001	decimal
X'F1'	241	1111 0001	hexadecimal
X'101'	257	1 0000 0001	hexadecimal
B'1111'	15	0000 1111	binary
B'11110001'	241	1111 0001	binary
C'1'	241	1111 0001	char. value
C'A'	193	1100 0001	char. value

Self-defining terms, constants, literals

The utilization of self-defining terms differs from that of data constants and literals:

– If data constants or literals are specified in instruction statement operands, their addresses are assembled into the instruction statement.
– If, on the other hand, a self-defining term is used in an instruction statement, its value is assembled into the instruction statement.

Example

```
Name        | Operation | Operands
------------+-----------+-----------------------
FIELD       | DS        | CL3
CONST       | DC        | C'ABC'
*           |           |
            | MVI       | FIELD,X'FF'          (01)
            | MVC       | FIELD,CONST          (02)
            | MVC       | FIELD,=C'ABC'        (03)
```

(01) The value of the direct operand is to be transferred to FIELD. FF is assembled into the generated machine instruction.
(02) The contents of CONST are to be transferred to FIELD. The address of CONST is assembled into the generated machine instruction.
(03) The literal C'ABC' is to be moved to FIELD. The address of the literal is assembled into the generated machine instruction.

Decimal self-defining terms

A decimal self-defining term is an unsigned decimal number, represented by a series of decimal numbers, in which there may also be leading zeroes.

A decimal self-defining term is assembled by the assembler into its binary equivalent.

Restrictions on the value depend on the purpose of its utilization:

– A decimal self-defining term may under no circumstances consist of more than ten digits or exceed the value 2^{31}-1.
– A decimal self-defining term which is to denote a general-purpose register must have a value of 0 to 15.
– A value which represents a displacement may not exceed the highest main memory address.

Hexadecimal self-defining terms

A hexadecimal self-defining term is an unsigned hexadecimal number represented as a series of hexadecimal digits. These must be enclosed in single quotes and immediately follow the letter X.

Each hexadecimal digit is converted by the assembler into a binary value of 4 bits. This means that a hexadecimal self-defining term which is to represent an 8 bit mask is made up of two hexadecimal digits.

The maximum permissible hexadecimal value is X'FFFFFFFF'.

Binary self-defining terms

A binary self-defining term is a series of unsigned binary digits, used to represent any required binary pattern. The series of digits must be enclosed in single quotes and directly follow the letter B.

A binary self-defining term may contain up to 32 binary digits. The value is padded to full bytes. To do this, the high-order byte is padded to the left with binary zeroes.

Binary self-defining terms are mainly used to represent the bit pattern of masks or in logical instructions.

Example

```
Name    | Operation | Operand
--------|-----------|-----------------
MASK    | EQU       | B'10101101'
ALPHA   | TM        | GAMMA,MASK
```

The binary self-defining term is used as a mask in a TM instruction. The contents of GAMMA are to be compared bit by bit with the bit pattern of the mask.

Character self-defining terms

A character self-defining term can be used to represent printable characters. A character self-defining term consists of one to four characters, enclosed in single quotes. It can be immediately preceded by the letter C.

All letters, digits and special characters may be used in character self-defining terms. The following rules apply to the use of single quotes (') and ampersands (&):

- The single quote (') is used as a syntax character in the assembly language, and the ampersand (&) as a syntax character in the macro language. Therefore, for each single quote or ampersand to be used in a character self-defining term, two single quotes or ampersands must be typed in. The two single quotes or ampersands are assembled as one single quote or one ampersand.

Examples

A'$	must be typed in as	C'A''$'
'A'	must be typed in as	C'''A'''

2.5.2.3 Location counter reference

Using an asterisk (*) as an element of an expression, the current value of the location counter may be referenced at any point in the source program.

The asterisk is assigned the current value of the location counter, i.e. the location counter of the instruction statement in which the asterisk is used.

The location counter reference can be used in all machine instructions and in the assembler instructions DC, DS, EQU, ORG and USING, which permit the asterisk in the operand entry. In addition, the location counter can be referenced in address constants and in literals specified as address constants (see section 4.2, DC instruction, for use of the asterisk in address constants with a duplication factor). One special case is the use of the asterisk in Format 2 of the USING instruction (see 4.2). Here it denotes the start address of the memory area for the dummy register vector.

The maximum value of the location counter is 2^{24}-1. This corresponds to a module size of 16 MB.

Examples

Loc. counter (hex)	Source program statement	Value of *
LOCTN	SOURCE STATEMENT	
000100	NAME B *+8	NAME (=000100)
000104	B NAME+8	
000108 <——	Target address of both branch instructions	
000120	CONSTANT DC A(*)	CONSTANT (=000120)
000134	ALPHA L R5,=A(*)	ALPHA (=000134)

2.5.2.4 Length attribute reference

The length attribute of a name is referenced by inserting an L’ directly in front of the name. The assembler replaces this expression with the implicit length of the name.

The length attribute of * (L’*) is the same as the length of the instruction statement in which the reference name occurs. An exception is the instruction EQU * without the length operand (see section 4.2, EQU instruction). Here the length attribute is 1.

Example

In the example, a character constant is positioned on the high or low-order byte of a field using the length attribute.

```
Name     | Operation | Operand
---------+-----------+-------------------------
A1       | DS        | CL8                      (01)
B2       | DC        | CL2’AB’                  (02)
*        |           |
HIORD    | MVC       | A1(L’B2),B2              (03)
LOORD    | MVC       | A1+L’A1-L’B2(L’B2),B2    (04)
```

(01) A1 denotes a memory area of 8 bytes, and has a length attribute of 8.

(02) B2 indicates a character constant of 2 bytes, and has a length attribute of 2.

(03) The instruction with the address HIORD moves the contents of B2 to the two high-order bytes of A1. The value L’B2 specifies the required length. When the instruction is assembled, the length is put into the appropriate field in the instruction statement.

(04) The instruction with the address LOORD moves the contents of B2 to the two low-order bytes of A1. The arithmetic expression A1+L’A1-L’B2 gives the seventh byte of A1 as the address. The length of A1 is added to the start address of A1, and the length of B2 subtracted from it. The contents of B2 are therefore moved to the seventh and eight byte of field A1. L’B2 also supplies the length specification required for the instruction.

2.5.3 Literals

Literals can be used to define data, e.g. numeric values, addresses or printable characters, without specifying DC instructions. Literals may replace the second operand of machine instructions.

Each literal is allocated a storage area in the literal pool by the assembler. The address for the literal in the literal pool is entered in generated machine instruction code.

Format of a literal: $=[dup]type[Ln_1][Sn_2][En_3]\left\{\begin{matrix}'chrcon' \\ 'datcon[,...]' \\ (adrcon[,...])\end{matrix}\right\}$

dup — Duplication factor
decimal self-defining term or positive absolute parenthesized expression, value range: 0 to 2^{24}-1

type — Literal type
a single letter, may be dropped for character constants

examples of literals:		
	C	character constant
	B	binary constant
	P, Z	decimal constants
	X	hexadecimal constants
	F, H	fixed-point constants
	E, D, L	floating-point constants
	A, Y, V	address constants

Ln_1 — Length modifier;
n_1 is a decimal self-defining term or a positive absolute expression in parentheses

Sn_2 — Scale modifier
En_3 — Exponent modifier

n_2 or n_3 is a positive or negative decimal self-defining term, respectively, or an absolute parenthesized expression.

chrcon — Value of the character constants
datcon — Value of the decimal, hexadecimal, binary, fixed-point, and floating-point constants
adrcon — Value of the address constants

Rules

- A literal begins with an equals sign (=). For the notation of a literal, the same rules apply as for an operand of a DC instruction (see section 4.2, DC instruction, especially the sections on "Modifiers" and "Types of constants").
- Literals may be used in all machine instructions in which the operand is a relocatable expression that is to be used for read accesses only.
  Literals may not be used in machine instructions that contain a relocatable expression as an operand to be used for write accesses.
- Literals may not be used in assembler instructions.
- A literal may not be an element of an expression.
- The duplication factor in a literal may not be zero.
- Address constants of type Q and S may not be used in literals.

Note

Whether the type of literal specified in a machine instruction corresponds with the operation code of the instruction is not checked.

Literal pool

The assembler determines the values of the literals and stores them in a specific area of memory, the literal pool.

The literal pool is usually at the end of the first control section. In that case the literal pool is printed after the END instruction in the assembler listing. Several literal pools may be defined using the assembler instruction LTORG (see section 4.2), and any desired position in the program selected for them.

The same literals are stored only once in the literal pool, unless these involve constants with location counter reference.

Examples

```
Name        | Operation | Operand                  |
------------|-----------|--------------------------|-------------------
AREA1       | DS        | 3CL4                     } these definitions apply
AREA2       | DS        | PL3                      } to all the following
HW          | DS        | H                        } instructions
            | .         |
            | .         |
            | MVC       | AREA1,=3CL4'ABCD'        (01)
            | UNPK      | AREA2,=P'352'            (02)
            | MVC       | HW,=H'80'                (03)
            | IC        | 5,=X'FF'                 (04)
            | LM        | 4,6,=A(AREA1,HW,*)       (05)
```

(01) C1C2C3C4C1C2C3C4C1C2C3C4 is moved to field AREA1.
(02) AREA2 contains the unpacked value F3F5C2.
(03) The value 0050 is moved to field HW.
(04) The IC instruction loads the value FF into the low-order byte of register 5.
(05) The addresses of AREA1 and HW and the current value of the location counter are loaded into registers 4, 5, and 6.

2.6 Remarks entry

The remarks entry contains explanations on the program which are to be included in the assembler listing.

All valid characters, including space characters, may be used in remarks (see section 2.1, "Character set").

The remarks entry must be separated from the operand entry by at least one space character.
Should the entry be continued in an additional line, there must be a continuation character in the continue column.

In instruction statements which are to contain remarks but no operands, a space-comma-space must precede the remarks entry.

Example

```
Name          | Operation | Operand
--------------|-----------|-----------------------
              | CSECT     | ␣,␣COMMENT
              | .         |
              | .         |
              | MVC       | FIELD1,FIELD2␣COMMENT
              | .         |
              | .         |
              | END       | ␣,␣COMMENT
```

2.7 Continuation character

The first column after the end column is the continuation character column. This column contains the continuation character. The default for the end column is column 71. The continuation character must be set if an instruction statement is to be continued into another line. Any character, apart from the space character, may be used for this purpose.

There must be no space character in front of the continuation character.

If the continuation character column contains a space, it will be assumed that the instruction statement has been terminated.

The instruction statement must be continued in the continue column of the next line:

- If the text in the continue column starts with a character which is not a space character, it is interpreted as a continuation of the current entry.
- If there is a space character in the continue column, the text which follows is evaluated as the next entry.

Any number of continuation lines are allowed per instruction statement.

3 Addressing, program sectioning and program linking

3.1 Addressing

The address designates a storage area. It consists of the contents of a base address register, to which the displacement is added (base/displacement addressing). For RX instructions, the contents of an index register may also be added.

There are two options for specifying addresses in a source program:

Non-symbols

In this instance, the base address register and displacement are clearly identified (see "Assembler Instructions" reference manual [3]).

Symbols

A symbol is first converted by the assembler into base/displacement form.
A symbol is a name or is generated via variable symbols or via a concatenation of variable symbols and alphanumeric characters.

Use of symbols

To enable the assembler to convert symbols into their non-symbolic form, the programmer must

- inform the assembler by means of a USING instruction of the general-purpose registers to be used as base registers,
- specify in the USING instruction the value assigned to each of these registers as base address, and
- at program execution time, load each of the base registers with the specified value.

On assembly, the symbol is converted by the assembler into its non-symbolic form and then converted into the object code.

If symbols are used, a program must contain a separate USING instruction for each control section.

Base address registers assigned with the USING instruction can be released for other purposes by means of the DROP instruction.

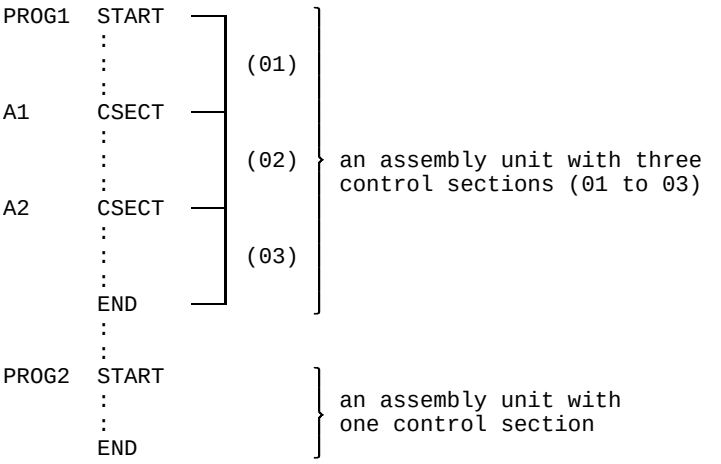
3.2 Program sectioning

The text of an assembler source program consists of one or more assembly units. An assembly unit usually begins with a START or CSECT instruction and is terminated with an END instruction. It is mostly designated as "a program".
Each assembly unit is assembled into an object module.

An assembly unit may consist of one or more control sections, which are assembled as parts of an object module.

The corresponding instructions enable the linkage editor to link one or more object modules into one executable program.

Example



It is generally necessary to refer to data that has been defined in another program section or to branch into another program section. To this end, intercommunication between the program sections must be achieved.

- Every individual program section accessed must be symbolically addressable (see section 3.1.1, "Symbols").
- The two or more assembly units which are to be linked must be linked symbolically (see section 3.2.2, "Symbolic program linking").

3.3 Control sections

A control section is the smallest possible program section which may be relocated independently of the others on loading, without affecting the operating logic of the program. A maximum of 2^{15}-1 control sections may be contained in an assembly unit.

There are executable control sections which are converted into object code, and reference control sections which are not converted into object code. They are used to describe data which can be referred to from executable control sections.

Location counter in control sections

Parts of different control sections may occur in an assembly unit in mixed sequence, as the assembler keeps a separate location counter for each control section.

The location counter of reference control sections has an initial value of zero, i.e. the address values within a control section are relative to the start of the control section.

Executable control sections are allocated storage areas one after the other, in the order in which they occur in the source program text. Each successive control section starts at the next free doubleword boundary, unless the PAGE attribute has been specified (see section 3.3.3, "Control section attributes").

The maximum value of the location counter in a control section, or the maximum value of the entire location counter of all control sections is 2^{24}-1.

3.3.1 Executable control sections

An executable control section (**c**ode **sect**ion) is initiated by a START or CSECT instruction, and converted into the object code.

First control section

The first control section in an assembly unit can be indicated by the START instruction with which a preliminary start address of the program may be specified (see section 4.2, START instruction).

The assembler creates the literal pool at the end of first control section, provided that the position of the pool has not been otherwise specified via the LTORG instruction.

If the first program section is to be continued at another point, a CSECT instruction with the same name entry must be used for this.

Other executable control sections

If other executable control sections are to appear after the first one in an assembly unit, these must be initiated via a CSECT instruction. These control sections may be continued with a CSECT instruction of the same name at another point (see section 4.2, CSECT instruction).

3.3.2 Reference control sections

No object code is generated for reference control sections. They are used to describe data which is to be accessed by executable control sections. They are initiated via a DSECT, XDSEC, COM or DXD instruction.

Dummy section

A dummy section describes a data format which is to be projected onto a memory area like a "template". The start of this data structure is denoted by a DSECT instruction. The names defined in a dummy section correspond with the format elements.

The data format is projected onto a memory area as follows:

- The address of the required memory area must be loaded into a register during the program run.
- This register must be defined in the program as base address register for the dummy section via a USING instruction. In this way, the data format is always projected onto the memory area whose address has been loaded into the base address register.

If names defined in a dummy section are used in machine instructions, they are replaced during the program run by the data from the memory location onto which the data format was projected..

External dummy section

An external dummy section is indicated by an XDSEC instruction. The same rules apply to its application as are valid for a dummy section (see under), with the following exception:

To define an external dummy section (XDSEC D), external information is transferred to the linkage editor, which uses this information to attempt to satisfy the external information specified for the references of an external dummy section (XDSEC R) in another assembly unit. The location counter is set to zero in the reference of an external dummy section (XDSEC R), and remains at zero for the entire section. The actual location counter of a name in a reference of an external dummy section is entered by the linkage editor in the instruction in which this name is used.

In this way, the evaluation of the location counter for names of an XDSEC R is relocated from the assembly level to the linkage level. The advantage of this is that in altering an XDSEC D symbol, it is only necessary to reassemble the modules which use the names to be modified, and not all the modules which use this XDSEC.

Note

The dynamic linking loader DLL (in use up to BS2000 V9.5) does not support the use of external dummy sections, i.e. programs must be linked with TSOSLNK, and no dynamic loading mechanism can be used. The dynamic binder loader DBL, introduced as of BS2000 V10.0, does support the use of external dummy sections and dynamic loading.

Common control section

A common control section is a memory area which can be accessed from independently assembled assembly units, which are linked and loaded as one program. It is identified by a COM instruction.

Using DS and DC instructions, a common control section can be broken down into subfields, defined relative to the start of the section. In other words, the structure of the common control section is predefined. Constants defined with DC instructions in a common control section are not assembled. Their fields can be loaded with data only at program execution.

A common control section must be addressable in each assembly unit from which it is to be accessed. For communication between assembly units and a common control section, the common section must be defined identically in each.

Dummy registers

Dummy registers are memory areas which may be accessed from several assembly units. They are used as work areas and for intercommunication between different assembly units. The dummy registers may be defined at any required position in an assembly unit, and need not appear in ascending and/or related order.

The assembler calculates the alignment and length of the dummy register and passes this information on to the linkage editor via ESD entries. During the linkage procedure, all dummy registers are combined from all the modules to be linked into a so-called "dummy register vector". At the same time, the linkage editor defines alignment, length and storage of the individual dummy registers.
When linking, the strongest attributes in the same dummy registers are used as the final valid attributes. For example, should a dummy register be aligned on a halfword boundary in one CSECT, and on a fullword boundary in another, this dummy register would be aligned on the fullword boundary after linkage.

The dummy register vector combined by the linkage editor defines only the structure of the dummy registers. The memory area for this must be loaded explicitly in an executable control section.

The length of this dummy register vector can be entered by the linkage editor in a word which must be reserved for this purpose using a CXD instruction.

Compared to common control sections and external dummy sections, the use of dummy registers has the following advantages:

1. Assume CSECT1 with PSREG1 and PSREG2
   CSECT2 with PSREG3 and PSREG4

   Both CSECTs are to be combined into one program.

   Within a CSECT, only the dummy register defined in this exact CSECT may be accessed, i.e. in CSECT1 only PSREG1 and PSREG2. PSREG3 and PSREG4 cannot be overwritten, either intentionally or unintentionally.

2. If the number or the data description of dummy registers are changed in a module, only this module must be reassembled and the entire program relinked. If, on the other hand, the data description of a common control section or external dummy section is altered, all modules affected by a change to the contents or length need to be altered and reassembled.

Note

The DLL (in use up to BS2000 V9.5) does not support the use of dummy registers, i.e. programs must be linked with TSOSLNK, and no dynamic load mechanism may be used. The BINDER linkage editor, introduced as of BS2000 V10.0, does support the use of dummy registers and dynamic loading. The BINDER, introduced as of BS2000 V10.0, does not support the use of external dummy sections and dynamic loading.

Dummy registers can be addressed in two ways:

– via a base address register

  The instruction USING *PRV,8 (see section 4.2, USING instruction, format 2 and Appendix 11.4, example 2) causes all addresses concerning dummy registers to use register 8 as the base register, regardless of any other valid base register. Register 8 must then be loaded with the start address of the memory area for the dummy register vector.

– via Q-type constants (see Appendix 11.4, example 1)

  At program runtime, the Q-type constants contain the offset of the dummy register at the start of the dummy register vector. In other words, this value must be added at the start of the memory area of the dummy register vector in order to be able to access the dummy register.

Dummy registers can be defined in two ways (see examples in Appendix 11.4):

– With the DXD instruction;

– With the DSECT instruction, using Q-type constants;
  this defines the entire dummy section as a dummy register. The DSECT structure can be used for accesses within this dummy register.

3.3.3 Control section attributes

To facilitate dynamic linking and loading, it is often necessary to give all data and instruction statements within a control section certain characteristics. The following attributes may be specified (in any order) in the operand field of START, CSECT and COM instructions:

PUBLIC	This attribute signifies that the section contains data or instruction statements (shareable).
READ	This attribute signifies that the section cannot be modified, i.e. is write-protected (read only).
PRVLGD	This attribute indicates that the section has been assigned a protection key and can therefore be accessed by privileged system routines only.
PAGE	This attribute indicates that the section is to begin on a page boundary which is a multiple of 4096.
RESIDENT	This attribute signifies that the specified section is loaded in memory and is resident there.

Attributes may be specified singly or in combinations. The final set of attributes in a control section is determined by combining all the characteristics. Attributes may appear in the instruction which defines the beginning of the control section. Attributes for instructions of the same name need not be repeated, although they may be for documentation purposes.

If no attributes are specified, the control section is regarded as private, modifiable and aligned on a doubleword boundary. Information regarding the characteristics of each control section are stored in the object module.

Addressing mode and load attribute

These two attributes are assigned to a control section with the AMODE or RMODE instruction.

AMODE	Via this instruction, a control section is assigned a software addressing mode, which in turn designates a hardware addressing mode that the control section awaits for its execution.
RMODE	This instruction assigns a load attribute to a control section. This specifies the area of the address space (over or under 16 megabytes) in which the program must or can be loaded.

3.4 Symbolic program linking

Symbolic program linking enables symbols defined in one assembly unit to be accessed from another unit. To do this, the assembler requires appropriate information, which it passes on to the linkage editor via ESD entries. The linkage editor replaces these symbolic references with actual addresses prior to or during loading.

A symbol which is to be accessed by another assembly unit must be identified to the assembler and the linkage editor via the ENTRY instruction. It is thereby defined as a symbol of an entry point.

In an assembly unit where symbols defined in another unit are used, these must be identified via the EXTRN or WXTRN instruction. Since these symbols are defined in another assembly unit, the assembler assigns a provisional value 0 and length attribute 1.
In order to access the symbol, a base register must be provided in the assembly unit which uses the EXTRN address. The value of the addresses must be loaded into the base register via an A-type constant (see section 4.2, DC instruction).

Another method of symbolic linking is the use of V-type constants (see section 4.2, DC instruction). These constants are regarded as indirect linkage points, generated from an externally defined symbol. Here, the symbol must not be identified using the EXTRN instruction.
V-type constants may be used for branching into other assembly units, but not for references to data in other assembly units.

An example of linking two independent assembly units is provided under the description of the EXTRN instruction (see section 4.2).

4 Assembler instructions

4.1 General

As opposed to machine instructions which permit the central processing unit to execute certain operations during the program run, assembler instructions permit the assembler to carry out certain operations during assembly. They are used to control assembly and to handle auxiliary functions.

Programming notes

1. No literals may be used as operands in assembler instructions.
2. If assembler instructions are used as model statements in a macro definition, the name, operation and operand entries can be generated using **variable symbols** (see section 5.1.2, "Format of the macro definition", and chapter 6, "Variable symbols").
3. In some instructions, a sequence symbol can be specified in the name entry. A sequence symbol in the name entry identifies the instruction as a branch destination in terms of macro language (see section 5.3.1, "Sequence symbols"). A sequence symbol in the name entry cannot be generated with variable symbols.

Assembler instructions can be divided into the following categories:

Allocation of values and attributes

EQU	equate
OPSYN	assign mnemonic operation code

Definition of data areas

DC	define constants
DS	reserve memory area
CXD	reserve memory area for the length of the dummy register vector.

Base register statements

USING	assign base address register
DROP	release base address register
STACK	save USING or PRINT status
UNSTK	restore USING or PRINT status

Program sectioning, program linking, control section attributes

START	define program start
CSECT	define control section
DSECT	define dummy section
XDSEC	define external dummy section
COM	define common control section
DXD	define dummy register
END	end of assembly
AMODE	allocate addressing mode
RMODE	allocate load attribute
ENTRY	identify ENTRY address
EXTRN	identify EXTRN address
WXTRN	identify conditional EXTRN address

Input control

ICTL	control input format
COPY	copy source program text from library element

Output to object module

ORG	set location counter
LTORG	define literal pool
CNOP	set no operation
PUNCH	copy text into object module
REPRO	copy continuation line into object module

Listing control

Listing control statements identify the particulars in the assembler listing. They are concerned only with the assembler listing, and do not generate any instructions or constants in the source program.

TITLE	listing heading
SPACE	line feed
EJECT	page feed
PRINT	control listing contents
STACK	save USING or PRINT status
UNSTK	restore USING or PRINT status

Note

The number of lines per page in the assembler listing cannot be affected by any of these statements. It must be controlled using the appropriate option (see "ASSEMBH User Guide" [1]).

4.2 Description of instructions

AMODE Assign addressing mode

Function

The AMODE instruction allocates an addressing mode to a control section.

Format

Name	Operation	Operands
[{name / .sym}]	AMODE	{24 / 31 / ANY}

name Name
.sym Sequence symbol

Description

name refers to a control section with the same name, and must correspond to the name of a START, CSECT or COM instruction.

If the name field is blank, the AMODE instruction refers to an unnamed control section.

.sym A sequence symbol is the same as a blank name field.

24 The control section is assigned the 24-bit addressing mode.

31 The control section is assigned the 31-bit addressing mode.

ANY The control section can be executed in both the 24-bit and the 31-bit addressing mode.

The "addressing mode" is described in section 3.2.1.3 and in the manual "Introductory Guide to XS Programming" [7].

Information regarding the addressing mode of a control section is shown in the ESD entry.

Programming notes

1. The AMODE instruction may appear at any point in the source program. The source program may contain as many AMODE instructions as required, but a specified name may only appear once.
2. No AMODE instruction may be set for an unknown common control section (see section 4.2, "COM instruction").
3. The addressing mode assigned to a control section is also transferred to the ENTRY name of this control section.
4. If no special functions are to be introduced, a control section should be assigned the attributes AMODE = ANY and RMODE = ANY.

Combinations of AMODE and RMODE

If the AMODE instruction is set for a control section, the following combinations with the RMODE instruction are possible for the same control section:

AMODE 24 with RMODE 24
AMODE 31 with RMODE 24 or RMODE ANY
AMODE ANY with RMODE 24 or RMODE ANY

Default

The following defaults apply if AMODE and/or RMODE have not been set for a control section:

Specified	Default
neither AMODE nor RMODE	AMODE 24 and RMODE 24
AMODE 24	RMODE 24
AMODE 31	RMODE 24
AMODE ANY	RMODE 24
RMODE 24	AMODE 24
RMODE ANY	AMODE 31

Table 4-1 Defaults for AMODE or RMODE

>>>>> See also RMODE instruction

CNOP Set no operation

Function

The location counter for the next instruction statement can be aligned on specific bytes in a word or halfword using the CNOP instruction.

Format

Name	Operation	Operands
[{name / .sym}]	CNOP	b,w

name Name
.sym Sequence symbol
b Absolute expression, possible values: 0,2,4,6
w Absolute expression, possible values: 4,8

Description

b specifies the byte in a word or doubleword to which the location counter is to be set.

w specifies whether the byte specified in b is to be in a word (w=4) or doubleword (w=8).

If an expression is specified for b or w, the names used need not be defined beforehand.

The following table shows the possible combinations of b and w and their meanings.

b,w	meaning
0,4	start of a word
2,4	middle of a word (second halfword)
0,8	start of a doubleword
2,8	second halfword of a doubleword
4,8	middle (third halfword or second word) of a doubleword
6,8	fourth halfword of a doubleword

Table 4-2 Permissible combinations of operands in the CNOP instruction

The table below shows the position of a word, or doubleword, which denotes each of these pairs of operands.

Doubleword							
Word				Word			
Halfword		Halfword		Halfword		Halfword	
Byte	Byte	Byte	Byte	Byte	Byte	Byte	Byte
0,4 0,8		2,4 2,8		0,4 4,8		2,4 6,8	

Table 4-3 Alignment of the location counter with the CNOP instruction

Programming notes

1. If the desired alignment requires the location counter to be incremented, one to three no-operations are generated using the CNOP instruction (see BCR and NOPR, "Assembler Instructions" reference manual [3]). These occupy two bytes each.
2. If the location counter is already aligned as desired, the CNOP instruction is not used to generate no-operations.
3. The name of a CNOP instruction is given the value of the location counter **prior to** any incrementation required by no-operations. The CNOP instruction itself is aligned on a halfword boundary.

COM Define common control section

Function

The COM instruction identifies the beginning or continuation of a common control section and reserves memory area for it. A common control section can be accessed from several assembly units.

Format

Name	Operation	Operands
[{name / .sym}]	COM	[type[,...]]

name Name
.sym Sequence symbol
type Attribute identification for control sections (see section 3.3.3, "Control section attributes")

Description

The name entry identifies the name of the common control section.

A COM instruction without a name identifies an unnamed common control section. The beginning of an unnamed COM section is logged in the ESD and XREF listing (see "ASSEMBH User Guide" [1]).

The start address of a common control section is always aligned on a doubleword boundary.

A common control section is given its own location counter with the initial value zero.

Programming notes

1. Using DS or DC instructions, a common control section can be broken down into subfields, which are then defined relative to its start.
2. Machine instructions or constants specified in a common control section are not assembled. The fields of a common control section can only be loaded at program execution (see example).
3. Several COM statements with the same name may appear in a program. The first designates the beginning, and the others the continuation of the common control section.
4. A common control section which is to be accessed from 2 assembly units must be identically defined and addressable in each of the two units.
5. No AMODE or RMODE instructions may be set for an unnamed control section.

Example

```
Name      | Operation | Operands
----------|-----------|-----------------------------
PR0G      | START     |
R2        | EQU       | 2
          | .         |
          | .         |
          | L         | R2,=A(HCOM)            (01)
          | USING     | HCOM,R2                (02)
          | MVC       | COM2,=C'12345'         (03)
          | .         |
          | .         |
HCOM      | COM       |                        ⎫
COM1      | DS        | F                      ⎬ (04)
COM2      | DS        | CL5                    ⎭
          | .         |
          | .         |
```

(01) Loads the start address of HCOM into register R2.
(02) Specifies HCOM as the base address for the common control section and allocates register R2 as its base address register.
(03) Enters data in the COM2 field.
(04) Defines the common control section.

>>>>> See also END, START, CSECT, DSECT and XDSEC instructions

COPY Copy source program text from library element

Function

The COPY statement copies the specified element from a library into a source program.

Format

Name	Operation	Operands
	COPY	name

Description

"name" is the name of the library element to be copied.

The COPY statement does not check whether the name is syntactically valid as regards library management.

The copied instructions of the library element are inserted after the COPY statement in the source program. The library or library section that has to be searched, can be specified by the user (see "LMS" User Guide [4] and "ASSEMBH User Guide" [1]).

If several elements with the same name exist in a library, the element with the highest version designation is used.

Programming notes

1. The nesting level of COPY statements can be specified via a //COMPILE statement option (see "ASSEMBH User Guide" [1]). Its default is 5, and it may be a maximum of 255.
2. COPY statements may appear both in the assembler source program and in macros.
   COPY statements within inner macro definitions are only expanded during definition processing of the inner macro.
   Copied instructions may in turn contain macro definitions.
3. The name of a COPY element may not be generated.
4. The copied text is interpreted according to any preceding ICTL instruction. As this may be unintentional, a warning is issued.

5. The copied text must not contain an ICTL instruction.
6. Logging of the copied statements may be controlled via a //COMPILE statement option (see "ASSEMBH User Guide" [1]) or with PRINT COPY or PRINT NOCOPY (see section 4.2, "PRINT instruction").

CSECT Define control section

Function

The CSECT instruction identifies the beginning or continuation of a control section.

Format

Name	Operation	Operands
[{name / .sym}]	CSECT	[type[,...]]

name Name
.sym Sequence symbol
type Attribute identification for control sections (see section 3.3.3, "Control section attributes")

Description

The name entry identifies the name of the control section which begins with or is continued with the CSECT instruction.

A CSECT instruction without a name identifies an unnamed control section. The beginning of an unnamed CSECT section is logged in the ESD and XREF listing (see "ASSEMBH User Guide" [1]).

The length attribute of name is 1.

Programming notes

1. All instruction statements which appear between a CSECT instruction and the next CSECT or DSECT instruction with other names belong to one control section.
2. Several CSECT instructions with the same name may appear in a program. The first designates the beginning, and the others the continuation of the control section. A second or further CSECT instruction without a name identifies the continuation of an unnamed control section.
3. If AID is used to debug an assembler source program, the CSECT instruction which designates the first control section must have a name. No LSD information is stored for assembler programs whose first control section is unnamed (see "AID - Debugging of ASSEMBH Programs" [2]).

>>>>> See also DSECT and XDSEC instructions

CXD Reserve memory area for the length of the dummy register vector

Function

The CXD instruction reserves a word in which the linkage editor enters the total length of the dummy register vector.

Format

Name	Operation	Operands
[{name / .sym}]	CXD	

name Name
.sym Sequence symbol

Description

The value of the name entry is the address of the memory area in which the linkage editor enters the total length of the dummy register vector.

This memory area has a length attribute of 4 and must be aligned on a fullword boundary.

Programming notes

The CXD instruction may appear at any desired point in the source program

>>>>> See also DXD instruction

DC Define constants

Function

The DC instruction defines constants in memory.

Format

Name	Operation	Operands
[{name / .sym}]	DC	{[dup]type[Ln$_1$][Sn$_2$][En$_3$]{'chrcon' / 'datcon[,...]' / (adrcon[,...])}[,...]

name Name
.sym Sequence symbol
dup Duplication factor
decimal self-defining term or a positive absolute parenthesized expression, value range: 0 to 2^{24}-1
type Type of constant
a single letter, may be dropped for character constants
Ln_1 Length modifier
n_1 is a decimal self-defining term or a positive absolute parenthesized expression
Sn_2 Scale modifier
En_3 Exponent modifier
n_2 or n_3 is a positive or negative decimal self-defining term, respectively, or an absolute parenthesized expression.
chrcon Value of character constants
datcon Value of decimal, hexadecimal, binary, fixed-point and floating-point constants
adrcon Value of address constants

A separate constant is generated for each operand in the DC instruction, or for each value defined within an operand.

Description

name is the name of the constant or name of the first of several constants.
The value of the name is the address of the high-order byte of the first constant.

The length attribute of the name is the same as the length of the constant explicitly defined in the length modifier. If no length modifier has been specified, the length attribute is the same as the implied length of the constant.

If there is more than one value or operand, the length attribute of "name" is the length in bytes of the first defined constant.
The length of the first constant must be added to "name" in order to access the other constants.

dup specifies how often a constant is to be generated.
If an expression is specified for dup, the names used in this expression need not have been defined previously.

A **duplication factor with the value zero** is permitted and has the following effect: no value is assembled, but the constant is aligned according to its type.

type specifies the type of constant defined. If no length modifier is specified, the type defines the alignment of the constant in memory and its length.

possible types of constant:

C	character constant
B	binary constant
P, Z	decimal constant
X	hexadecimal constant
F, H	fixed-point constant
E, D, L	floating-point constant
A, Y, S, V, Q	address constant

(see Types of Constants).

Ln_1, Sn_2, En_3
See "modifiers" in the DC instruction

chrcon, datcon, adrcon
chrcon, datcon and adrcon are the values of the constants.

If several values are specified, the attributes described for each separate value are applicable.

Alignment of constants

The alignment of constants, i.e. the incrementation of the location counter to a specific boundary, depends on the type of constant. If a length modifier is specified, no alignment is ever carried out.

If the operand contains more than one constant, then only the first constant is aligned.

Type	Alignment on
C	byte boundary
X	byte boundary
B	byte boundary
P	byte boundary
Z	byte boundary
F	fullword boundary
H	halfword boundary
E	fullword boundary
D	doubleword boundary
L	doubleword boundary
A	fullword boundary
Y	halfword boundary
S	halfword boundary
V	fullword boundary
Q	fullword boundary

Table 4-4 Alignment of DC constants

Padding and truncation of constants

If more space than is required is provided for the value of a constant, then the extra space is padded.

If insufficient space has been provided for a constant, it is truncated, and a part of the constant is lost.

Type	Pad	Truncate
C	to the right with space chars.(X'40')	to the right
X	to the left with binary zeros	to the left
B	to the left with binary zeros	to the left
P	to the left with binary zeros	to the left
Z	to the left with EBCDIC zeros (X'F0')	to the left
F	to the left as per sign bit	to the left
H	to the left as per sign bit	to the left
E	to the right with binary zeros	not applicable ⎫
D	to the right with binary zeros	not applicable ⎬ (01)
L	to the right with binary zeros	not applicable ⎭
A	to the left with binary zeros	to the left
Y	to the left with binary zeros	to the left
S	to the left with binary zeros	to the left
V	to the left with binary zeros	to the left
Q	to the left with binary zeros	to the left

(01) Floating-point constants are not truncated, but are reported as errors and not assembled.

Table 4-5 Padding and truncation of DC constants

Storage space

The storage space reserved per operand in a DC instruction is evaluated as follows:

```
length modifier x number of values x duplication factor
+ all bytes skipped for alignment.
```

If more than one operand is specified, the necessary storage space is calculated from the sum of the storage space for the individual operands.

Location counter reference

When a constant is aligned, the location counter is incremented to the appropriate boundary.

If an address constant refers to the location counter, the first byte of the constant is used as the value of the memory address, i.e. the value of the location counter changes from one constant to the next by the length of the constant if several address constants in a DC instruction refer to the location counter.

If a constant which refers to the location counter is specified with a duplication factor, the constant is duplicated with the new value of the location counter.

Examples

```
Name        | Operation | Operands
------------+-----------+-------------------
DUP         | EQU       | 2
            | .         |
            | .         |
CHRCON1     | DC        | C'ABC'            (01)
CHRCON2     | DC        | 2CL5'ABC'         (02)
CHRCON3     | DC        | (DUP)CL5'ABC'     same meaning as in CHRCON2
            | .         |
            | .         |
HEXCON1     | DC        | X'99'             (03)
HEXCON2     | DC        | X'99,F7D5,0'      (04)
HEXCON3     | DC        | XL3'A6F4E'        (05)
            | .         |
            | .         |
ADRCON1     | DC        | A(CHRCON1)        (06)
ADRCON2     | DC        | A(*+4096)         (07)
ADRCON3     | DC        | A(*+4096,*)       (08)
```

	Generated constants	Meaning
(01)	C1C2C3	length: 3 bytes
(02)	C1C2C34040C1C2C34040	2 constants, both padded to length 5
(03)	99	length: 1 byte
(04)	99F7D500	3 constants, length: 1 byte, 2 byte, 1 byte, address of the 2nd or 3rd constant: HEXCON2+1, or HEXCON2+3
(05)	0A6F4E	length: 3 bytes, constant padded to the left
(06)	Address of CHRCON1	length: 4 bytes
(07)	Address of the 1st byte of ADRCON2 + 4096	length: 4 bytes
(08)	Value of the first constant:	Address of the 1st byte of ADRCON3 + 4096,
	Value of the second constant:	Address of the 1st byte of ADRCON3 + 4; both constants have a length of 4 bytes.

Modifiers

The modifiers of a constant are the length in bytes which a constant is to receive, the scale modifier, and the exponent modifier.

Length modifier

The length modifier overwrites the implied length of a constant. It specifies the number of bytes reserved for a constant. From this, it determines whether a constant is padded, or its value truncated. If a length modifier is specified, the constant is not aligned.

Format of the length modifier Ln_1

n_1 is a decimal self-defining term or a positive absolute parenthesized expression

If an expression is specified for n_1, the names used need not have been defined previously.

n_1 may not exceed the maximum permissible value for the various types of constants.

Type	Implied length	Possible values for n_1 (byte)
C	(01)	1 to 256
X	(01)	1 to 256
B	(01)	1 to 256
P	(01)	1 to 16
Z	(01)	1 to 16
F	4	1 to 8
H	2	1 to 8
E	4	1 to 8
D	8	1 to 8
L	16	1 to 16
A	4	1 to 4
Y	2	1 or 2
S	2	2
V	4	3 or 4
Q	4	1 to 4

(01) The implied length is calculated according to the length of the defined constant.

Table 4-6 Length modifiers in DC constants

Scale modifier

The scale modifier may only be used in fixed- and floating-point constants. It defines the internal positional shift for a constant, i.e. the number of binary digit positions for fixed-point constants and hexadecimal digit positions for floating-point constants.

Format of the scale modifier Sn$_2$

n_2 is a positive or negative self-defining term or an absolute parenthesized expression

n_2 may contain an operational sign. If no sign is specified, a plus sign is assumed.

Type	possible values for n_2
F	-187 to +346
H	-187 to +346
E	0 to +14
D	0 to +14
L	0 to +28

Table 4-7 Highest values of scale modifiers in fixed- and floating-point constants

Scale modifier in fixed-point constants

Here the scale modifier specifies the power of two by which a constant is to be multiplied after its value has been converted into the internal binary notation, but before it is assembled into the appropriate position.

Multiplication of a binary number by the power of two causes the binary point to be moved away from its initial position and to the right, behind the last digit position.

The scale modifier therefore specifies the following:

If n_2 is positive:	the number of binary positions which the fractional part of the binary number is to occupy. The integer part of the constant is therefore shifted to the left here.
If n_2 is negative:	the number of binary positions to be deleted from the integer part of the binary number. The integer part of the constant is therefore shifted to the right here.

If positions are lost due to the scale modifier being specified or not being specified, the position on the farthest right of the binary number is rounded off (i.e. up for a number > 5; down for a number < 5).

If no scale modifier is specified in a fixed-point constant which contains comma positions, the positions behind the comma are lost.

Scale modifier in floating-point constants

Here, n_2 specifies the number of hexadecimal positions by which the mantissa in the binary notation of a floating-point constant is to be moved to the right. The first position of the mantissa is assumed to be directly behind the hexadecimal point (normalized floating-point constant) originally.

The scale modifier creates a unnormalized floating-point constant, i.e. the positions to the extreme left of the mantissa contain hexadecimal zeros.

n_2 must be positive in this instance.

If the mantissa is moved by the scale modifier, the characteristic of floating-point constant is corrected accordingly.

If positions are lost as a result of a scale modifier having been specified, the position on the extreme left of the lost part is rounded off.

Exponent modifier

The exponent modifier can only be used in fixed- and floating-point constants. It defines the decimal power by which the value of a constant is to be multiplied before being converted into its internal binary notation.

`Format of the exponent modifier` En_3

n_3 is a positive or negative self-defining term or an absolute parenthesized expression

n_3 may contain an operational sign. If no sign is specified, a plus sign is assumed. If an expression is specified for n_3, the names used need not have been defined previously.

The permitted range for n_3 is -85 to +75.

Note

En_3 should not be confused with the exponent which may be defined for a value in the datcon field. If both types of exponent definition exist in an operand, their values are added together before being converted into binary notation. This exponent total must also lie within the permitted range of -85 to +75.

Types of constants

"type" in the DC instruction specifies the type of constant defined (see Table 4-7). With this as a basis, the assembler can interpret the constant and assemble it into the appropriate machine format. If no length modifier is specified, "type" defines the alignment of the constant in memory, and the storage space occupied by the constant.

Code	Type of Constant	Machine Format
C	character constant	8-bit code for each character
X	hexadecimal constant	4-bit code for each hexadecimal character
B	binary constant	binary format
F	fixed-point constant	binary fixed-point notation with sign, one word
H	fixed-point constant	binary fixed-point format with sign, one halfword
E	floating-point constant	abbreviated floating-point format, one word
D	floating-point constant	full floating-point format, one doubleword
L	floating-point constant	extended floating-point format, two doublewords
P	decimal constant	packed decimal format
Z	decimal constant	zoned decimal format
A	address constant	address value, one word
Y	address constant	address value, one halfword
S	address constant	base register and displacement value, one halfword
V	address constant	storage space reserved for external symbols; one word per address
Q	address constant	storage space reserved for the offset of a dummy register relating to the start of a dummy register vector

Table 4-8 Types of constants

The individual constants are described below.

Character constant C

The character constant can be used to specify character strings. Character constants may be formed using any printable characters. Each character specified in the chrcon field is assembled into one byte.

There is no alignment in memory.

The greatest length modifier allowed is 256.
If no length modifier is specified, the length of the constant corresponds to the number of characters in chrcon. If a length modifier is specified, the value of the constant is

– truncated to the right if the length modifier is too small, and
– padded to the right with space characters (X'40') if the length modifier is greater than the number of characters.

The following rules apply to the use of single quotes (') and ampersands (&):
The single quote (') is used as a syntax character in the assembly language, and the ampersand (&) as a syntax character in the macro language. Therefore, for each single quote or ampersand to be used in a character constant, two single quotes or ampersands must be entered. The two single quotes or ampersands are assembled as one single quote or ampersand.

Examples

```
Name      | Operation | Operands
----------|-----------|-------------------
CHRCON1   | DC        | C'ABC,DEF'          (01)
CHRCON2   | DC        | C'&& ABC,DEF'       (02)
CHRCON3   | DC        | 3CL4'12345'         (03)
```

	Generated constants	Meaning
(01)	C1C2C36BC4C5C6	Length: 7 bytes, comma is interpreted as a character.
(02)	5040C1C2C36BC4C5C6	Length: 9 bytes, the two ampersands are regarded as one character.
(03)	F1F2F3F4F1F2F3F4F1F2F3F4	Constant with duplication factor and length modifier; if the defined length is too short, the constant is truncated to the right.

Hexadecimal constant X

A hexadecimal constant consists of one or more hexadecimal digits. Each hexadecimal position defined in datcon is assembled into four bits. In an odd number of positions, the left four bits of the high-order byte are padded with hexadecimal zero.

The greatest length modifier is 256 (bytes).
If no length modifier is specified, the implied length of the constant is equal to the number of bytes in datcon. If a length modifier is specified, the constant is

- truncated to the left if length modifier is too small, and
- padded to the left with hexadecimal zeros, if the length modifier is greater than half the number of hexadecimal positions.

Examples

```
Name      | Operation | Operands
----------|-----------|-------------------
HEXCON1   | DC        | X'FF00FFFF'          (01)
HEXCON2   | DC        | 3XL2'BD8E7'          (02)
HEXCON3   | DC        | 3X'BD8E7'            (03)
```

	Generated Constants	Meaning
(01)	FF00FFFF	The constant generates the bit pattern of a word. HEXCON1 sets the first, third and fourth byte of a word to 1.
(02)	D8E7D8E7D8E7	Constant with duplication factor and length modifier; since the defined length is too short, the constant is truncated to the left.
(03)	0BD8E70BD8E70BD8E7	The length is calculated implicitly, and the constant is padded to the left with a hexadecimal zero.

Binary constant B

Using a binary constant, any desired bit pattern can be defined. It consists of a series of the binary digits 0 and 1.

The greatest possible length modifier is 256 bytes.
If no length factor is specified, the implied length of the constant is the number of bytes occupied by the constant. If the value of the constant does not occupy any integer number of bytes, it is padded to the left with binary zeros. If a length modifier is specified, the value of the constant is

- truncated to the left if the length modifier is too small, and
- padded to the left with binary zeros if the length modifier specified is greater than necessary for the defined bits.

Examples

```
Name      | Operation | Operands
----------|-----------|-------------------
BCON      | DC        | B'10101101'          (01)
BSHORT    | DC        | BL1'111100011'       (02)
BLONG1    | DC        | BL1'110'             (03)
BLONG2    | DC        | B'110'               (04)
```

	Generated constants	Meaning
(01)	AD	Length: 1 byte
(02)	E3	The constant is truncated to the left.
(03)	06	The constant is padded to the left with binary zeros.
(04)	06	Same effect as BLONG1, length is calculated implicitly.

Fixed-point constants F and H

Fixed-point constants define data which can be used in fixed-point instructions.

The value of a fixed-point constant is written as a decimal number which can be followed by a decimal exponent:

– The decimal number may be an integer, a fraction, or mixed, and have a positive or negative sign. If there is no sign, it is assumed to be positive. If there is no decimal point, the decimal number is interpreted as an integer.
– If a decimal exponent is specified as the value of the constant, it must follow immediately after the number. It is written as En, where n must be a decimal number which may have a sign. En is interpreted as an exponent to the base 10. The sum of the exponent and exponent modifier must not exceed the permitted range of -85 to +75.

Assembly of fixed-point constants

1. The defined number, multiplied by 10 to the power of the exponent, is converted into a binary number with sign. The sign is coded in the high-order bit.
   A negative number is represented as a twos complement with a sign bit set to 1.
2. This number is aligned according to the specified scale modifier. If a fractional or a mixed number is specified without a scale modifier, the fractional part is lost.
3. If necessary, the binary value is rounded off.
4. Once the constant is assembled, the duplication factor is evaluated.

Alignment, length, value range

The constant is aligned on a word or halfword boundary if no length modifier is specified.
The implied length for word constants (F) is four bytes, and for halfword constants (H) two bytes. With a length modifier, however, lengths of up to eight bytes may be specified for both types of constants. The following range of representable values for fixed-point constants is the result:

Length	Representable value range
8	-2^{63} to $2^{63}-1$
4	-2^{31} to $2^{31}-1$
2	-2^{15} to $2^{15}-1$
1	-2^{7} to $2^{7}-1$

Table 4-9 Representable value range for fixed-point constants

The value range is dependent on the implied or explicit length defined.
If the constant is smaller than its defined length, it is padded to the left as per the sign bit.
If the constant exceeds the range of values (which is possible as a result of its defined length), the sign is lost, and the constant is truncated to the left.

Examples

```
Name     | Operation | Operands
---------|-----------|-----------------
HCON1    | DC        | H'+12'          (01)
HCON2    | DC        | H'-12'          (02)
FCON1    | DC        | F'12.3'         (03)
FCON2    | DC        | FS8'12.3'       (04)
FCON3    | DC        | FS8'123'        (05)
```

	Generated constants	Meaning
(01)	000C	Length: 2 bytes
(02)	FFF4	The negative value is represented as a twos complement; the sign bit is on 1.
(03)	0000000C	Word constant; the fractional part is lost because no scale modifier was specified; rounding is downward.
(04)	00000C4D	The scale modifier reserves 1 byte for the fractional part, and the integer part of the constant is moved to the left; rounding is upward.
(05)	00007B00	The scale modifier reserves 1 byte for the fractional part even if none is contained in the value.

Floating-point constants E, D and L

Floating-point constants define data which may be used in floating-point instructions.

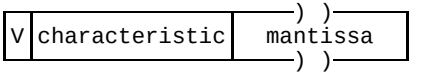
The value of a floating-point constant is written as a decimal number, which may be followed by an exponent:

– The decimal number may be an integer, a fraction, or mixed, and have a positive or negative sign. If there is no sign, the number is assumed to be positive. If there is no decimal point, the number is interpreted as an integer.
– If an exponent is specified for the number, it must be written in the form En, and follow it directly. "n" must be a decimal number which may have a sign. En is interpreted as an exponent to the base 10.
  The sum of the exponent and exponent modifier must not exceed the permissible range of -85 to +75.

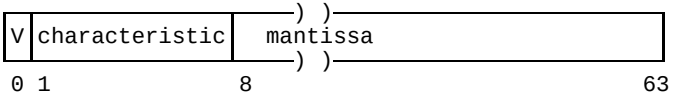
Machine format of floating-point constants

The machine format for a floating-point constant consists of two parts: the exponent part (characteristic), followed by the fractional part (mantissa). A sign bit indicates whether a positive or negative number has been defined (see "Assembler Instructions" reference manual [3]).

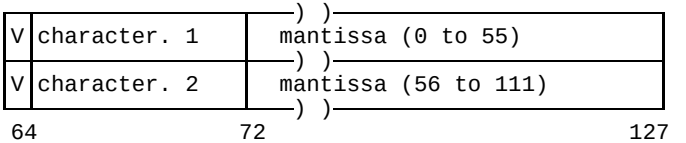
Abbrev. floating-point format (4 bytes):



Full floating-point format (8 bytes):



Extended floating-point format (16 bytes):



- The sign in the second doubleword is the same as in the first
- characteristic 2 = (characteristic 1 - 14)

Assembly of floating-point constants

A floating-point constant must be converted into a fraction prior to its conversion into machine code. The exponent is then converted into its binary equivalent, and the fraction converted into a binary number.

The assembler assembles a floating-point constant into its binary form, as explained in the following *example*:

The floating-point constant KON was defined as follows:

Name	Operation	Operands
KON	DC	E'-167.0'

1. $-167.0_{10} = -A7_{16}$ Fraction converted into a binary number
2. $-A7_{16} = (-0.A7*16^2)_{16}$ Normalization
3. Characteristic = $(\text{exponent} + 64 = 66)_{10} = 42_{16}$
4. The negative sign causes the high-order bit to be set to 1, which corresponds to an addition of 80_{16}:

   $80_{16} + 42_{16} = C2_{16}$
5. An E-type constant has an implied length of 4 bytes, i.e. the first byte contains the characteristic and sign; the 3 bytes on the right contain the mantissa:

   C2 A7 00 00 i.e. C2 = sign + characteristic
   A7 00 00 = mantissa
6. If a scale modifier was defined, hexadecimal zeros are added to the left of the normalized fractional part, and the characteristic is adjusted accordingly.
7. The fractional part is rounded off according to the implied or explicit length of the constant.
8. A negative fractional part is not represented as a twos complement, but in direct form with sign.
9. Once the constant is assembled, the duplication factor is evaluated.

Alignment, length, value range

E-type constants are aligned on word boundaries, D and L constants on double-word boundaries. If a length modifier is defined, there is no alignment.
The implied length for E-type constants is 4 bytes, for D-type constants 8 bytes, and for L-type constants 16 bytes. With a length modifier, up to 8 bytes may be specified for each length for E-type and D-type constants, and up to 16 bytes constants of type L.
The result is a range of values which can be assembled for the mantissa of floating-point constants:

Type	Mantissa Range of Values (exact)	Range of Values (approx.)
E	16^{-65} to $(1-16^{-6})*16^{63}$	$5{,}4*10^{-75}$ to $7{,}2*10^{75}$
D	16^{-65} to $(1-16^{-14})*16^{63}$	
L	16^{-65} to $(1-16^{-28})*16^{63}$	

Table 4-10 Representable mantissa range of values in floating-point constants

If the value specified for a constant does not lie within these ranges, the constant is not assembled and is flagged as an error.

Examples

```
Name      | Operation | Operands
----------|-----------|-----------------
ECON1     | DC        | E'167'            (01)
ECON2     | DC        | E'16.7'           (02)
ECON3     | DC        | ES2'16.7'         (03)
ECON4     | DC        | EE10'16.7'        (04)
ECON5     | DC        | E'16.7E10'        (05)
```

	Generated constants	Meaning
(01)	42A70000	Floating-point constants with a length of 4 bytes; the first byte contains the characteristic and sign, the 3 bytes on the right contain the mantissa.
(02)	4210B33	The characteristic remains the same; the mantissa is altered as a comma position has been specified.
(03)	440010B33	The scale modifier causes the fractional part to be shifted two hexadecimal positions to the right. The characteristic is altered, since the value of the exponent changes as a result of the shift.
(04)	4A26E1FA	Because of the exponent modifier, the value 16.7 is multiplied by 10^{10}, before it is converted.
(05)	4A26E1FA	Has the same result as ECON4. $16.7*10^{10}$ is assembled.

Decimal constants P and Z

Decimal constants define data which can be used in decimal instructions.

The value of a decimal constant is written as a decimal number, which may have a sign. If no sign is specified, the number is interpreted as positive. The decimal number may contain a decimal point. This is not taken into consideration when the constant is assembled into the internal format.

The greatest length modifier is 16 bytes.
If no length modifier is specified, the implied length of the constant corresponds to the number of bytes which it occupies.
If a length modifier is specified, the value of the constant is

- truncated to the left if the constant requires more bytes than specified in the length modifier, and
- padded to the left if the length modifier is greater than required. With Z-type constants, a decimal zero is used for padding, and with P-type constants, the bits of each byte added are set to zero.

Assembly of decimal constants

With packed decimal constants (P), each digit is assembled into its 4-bit binary equivalent. The sign appears in the 4 bits at the extreme right of the constant.
If an even number of packed decimal positions are defined, the left 4 bits of the byte on the extreme left are set to zero, and the 4 bits on the right contain the first position of the decimal number.
For unpacked decimal constants (Z), each digit is converted into its 8-bit long EBCDIC form. The sign is in the first 4 bits of the byte at the extreme right of the constant.

For both types of constant, a plus is stored as C, and a minus as D.

Examples

```
Name     | Operation | Operands
---------+-----------+-----------------
PCON1    | DC        | P'+153'          (01)
PCON2    | DC        | P'-153'          (02)
ZCON1    | DC        | Z'-153'          (03)
ZCON2    | DC        | Z'-1.53'         (04)
```

	Generated constants	Meaning
(01)	153C	length: 2 bytes
(02)	153D	length: 2 bytes
(03)	F1F5D3	length: 3 bytes
(04)	F1F5D3	length: 3 bytes, the decimal point is ignored.

Address constants

Address constants contain memory addresses. Address constants can be used, for example, to initialize a base register or to link control sections.

A-type and Y-type address constants

The value of an A- or Y-type constant defined in adrcon may be an absolute or a relocatable expression.

A-type constants are aligned on a fullword boundary and Y-type constants on a halfword boundary if no length modifier is specified.

The implied length of A-type constants is 4 bytes, of Y-type constants 2 bytes. With the length modifier, lengths of 1 to 4 bytes may be specified for A-type constants and 1 to 2 bytes for Y constants.

Assembly of A- and Y-type constants

– adrcon is an absolute expression:

  The value in adrcon is calculated to 32 bits, and then truncated to the left or padded to the left, as the length of the constant necessitates.

– adrcon is a relocatable expression:

The assembler uses the location counter as a provisional value. The actual value of the constant is only used when loading the program into the constant.

Programming notes

1. For 31-bit addressing, only 4-byte long A-type constants may be used.
2. A relocatable expression in an A- or Y-type constant may contain external names, and thus denote an address in an independently assembled program. In this address constant, the assembler enters the provisional value zero (see Example 03).
3. Y-type constants are frequently used in offset addressing in tables or data areas allocated by the executive macro REQM (see "Executive Macros" reference manual [6]).

Examples

```
Name      | Operation | Operands
----------+-----------+-----------------------
ACON1     | DC        | A(*+4096)              (01)
ACON2     | DC        | A(ACON1)               (02)
          | .         |                        |
          | .         |                        |
PROGRA    | START     |                        |
          | .         |                        |
          | EXTRN     | MARK                   |
          | .         |                        |
          | L         | 15,ACON                |
          | .         |                        |
ACON      | DC        | A(MARK)                |
          | .         |                        |
          | END       |                        } (03)
          | .         |                        |
          | .         |                        |
PROGRB    | START     |                        |
          | .         |                        |
          | ENTRY     | MARK                   |
          | .         |                        |
MARK      | EQU       | *                      |
          | .         |                        |
          | END       |                        |
```

	Value of the constant	Meaning
(01)	ACON1 location counter + 4096	The relocatable expression in the constant contains a reference to the location counter. The value of * is the ACON1 location counter.
(02)	ACON1 location counter	
(03)	Zero	Possible linking of two programs. The A-type constant in PROGRA is identified as an EXTRN address (branch address), in PROGRB as an ENTRY address (entry point). The branch destination is in PROGRB.

S-type address constants

S-type constants enable addresses to be filed in base-displacement form.

The value of an S-type constant may be specified as a symbol or a non-symbol A symbol is broken down by the assembler into base register and displacement value, while a non-symbol must be specified in the form (displacement(base register)).

An S-type constant is aligned on a halfword boundary, and its implied length is 2 bytes. The 4 bits on the extreme left of the assembled constant contain the number of the base register, and the remaining 12 bits contain the displacement value.

S-type constants may not be used in literals.

Programming notes

S-type constants are primarily used if a machine instruction code is not to be generated using a mnemonic operation code, but using DC constants.

Example

```
Name        Operation   Operands
            .
            .
R0          EQU         0
R1          EQU         1
R11         EQU         11
            .
            .
            BALR        R11,R0
            USING       *,R11
            .
            .
            L           R1,ADR                  (01)
            .
            .
            DC          X'58'                   ┐
            DC          X'10'                   } (02)
            DC          S(ADR)                  ┘
            TERM
            .
            .
ADR         DC          C'ABCD'
            END
```

	Generated object code	Meaning
(01)	58 10 B026	
(02)	58 10 B026	The string of DC constants generates an object code which corresponds to the L instruction.

V-type address constants

Using V-type constants, storage space can be reserved for entry points in another program. V-type constants may only be used as branch addresses, not for addressing external data.

The value of the constant is specified as a name. This name must not be identified using an EXTRN instruction, since it is automatically interpreted by the assembler as an external address (for the sake of compatibility, however, no flag will be output even if it is). Note: specifying a name in a V-type constant does not mean that it is reciprocally defined as an external symbol for this program.

The value of a V-type constant is zero until the program is loaded. The correct value of the address is inserted by the loader.

V-type constants are aligned on a fullword boundary if no length modifier is specified. The implied length of a V-type constant is 4 bytes. With the length modifier, a length of 3 or 4 bytes may be specified. If this is the case, the relative value of the constant may be truncated.

Programming notes

For 31-bit addressing, only V-type constants with a length of 4 bytes may be used.

Example

```
Name      | Operation | Operands
----------+-----------+-----------------
PROGRA    | START     |                  ⎫
          | .         |                  ⎪
          | L         | 15,VCON          ⎪
          | BALR      | 14,15            ⎪
          | .         |                  ⎪
VCON      | DC        | V(MARK)          ⎪
          | .         |                  ⎪
          | END       |                  ⎪
          | .         |                  ⎪  (01)
          | .         |                  ⎪
PROGRB    | START     |                  ⎪
          | ENTRY     | MARK             ⎪
          | .         |                  ⎪
          | USING     | MARK,15          ⎪
MARK      | LA        | ...              ⎪
          | .         |                  ⎪
          | .         |                  ⎪
          | BR        | 14               ⎪
          | .         |                  ⎪
          | END       |                  ⎭
```

(01) This example illustrates program linking achieved using a V-type constant. The name MARK may not be identified in the same program via an EXTRN instruction.

Q-type address constants

A Q-type constant is used to reserve storage space in which the offset of a dummy register is to be stored in the dummy register vector. In the dummy register vector, the dummy registers are one behind the other, in a sequence determined by the linkage editor. In other words, the Q-type constant contains the offset of the dummy register at the start of the dummy register vector.

The value defined in a Q-type constant is the name of a dummy register defined with DXD, or a DSECT also entered as a dummy register via referencing in a Q-type constant. This address is entered only by the linkage editor.

For a description on the use of Q-type constants, see section 3.2.1.2, "Dummy registers".

Note

Literals may not contain Q-type constants.

Example

```
Name        | Operation | Operands
------------|-----------|-----------------
DNAME       | DSECT     |
D1          | DS        | 3F
D2          | DS        | CL15
            | .         |
            | .         |
CNAME       | CSECT     |
            | .         |
            | .         |
QCON        | DC        | Q(DNAME)          (01)
            | END       |
```

(01) DNAME defines a DSECT. QCON contains the displacement from DNAME at the start of the dummy vector.

DS Reserve storage space

Function

The DS instruction reserves memory areas and allocates names to them.

Format

Name	Operation	Operands
[{name / .sym}]	DS	[dup]type[Ln][{'chrcon' / 'datcon'}]

name Name
.sym Sequence symbol
dup Duplication factor
decimal self-defining term or positive absolute parenthesized expression, value range: 0 to 2^{24}-1
type Type of reserved storage space, a single letter
Ln Length modifier
n is a decimal self-defining term or a positive absolute parenthesized expression
chrcon Value specification of character area
datcon Value specification for decimal, hexadecimal, binary, fixed-point and floating-point areas

Values specified in the operands of the DS instruction are not assembled into object code. They are only used, where necessary, to define implied length modifiers.

Note

The entire format of the DC instruction can be specified in the operand of the DS instruction. These extra specifications have no effect on the reserved area.

Description

name is the name of the reserved memory area. The value of the name is the address of the byte on the extreme left of the reserved area.

The length attribute of the name is the same as the length of the area explicitly defined in the length modifier. If no length modifier is specified, the length attribute is the same as the implied length of the area, which is dependent on the type specified.

dup specifies how often an area is to be reserved. If an expression is specified for dup, the names used in this expression need not have been defined previously.

A **duplication factor with the value zero** is allowed, and has the following effect: no memory area is reserved; the area is, however, aligned according to its type, and receives a length attribute.

type If no length modifier is specified, the type determines the alignment of the area in memory, and its length.

Possible Type Entries:		
	C	Character areas
	B	Binary areas
	P, Z	Decimal areas
	X	Hexadecimal areas
	F, H	Fixed-point areas
	E, D, L	Floating-point areas
	A, Y, S, V, Q	Address areas

Ln specifies the length in bytes of the area which is to be reserved, and overwrites the implied length.
If an expression is specified for n, the names used need not have been defined previously.

If a length modifier is specified, the corresponding area is not aligned.

Type	implied length	possible values for n (byte)
C	1	1 to 2^{24}-1
X	1	1 to 2^{24}-1
B	1	1 to 256
P	1	1 to 16
Z	1	1 to 16
F	4	1 to 8
H	2	1 to 8
E	4	1 to 8
D	8	1 to 8
L	16	1 to 16
A	4	1 to 4
Y	2	1 or 2
S	2	2
V	4	3 or 4
Q	4	1 to 4

Table 4-11 Length modifiers in the DS instruction

chrcon,datcon

chrcon and datcon are the values which may be specified in the operand.

If no length modifier is specified for types C, B, P, Z and X, the assembler calculates the length of the area to be reserved using the specified value.

If a value is specified for an area, it must be valid for the type of the corresponding area (see description of Types of Constants).

Alignment of memory areas

The alignment of memory areas, i.e. incrementation of the location counter to specific boundaries, depends on the type of area. If a length modifier is specified, no alignment is ever carried out.

If the operand contains more than one value specification, only the first value is aligned.

Type	Alignment on
C	byte boundary
X	byte boundary
B	byte boundary
P	byte boundary
Z	byte boundary
F	fullword boundary
H	halfword boundary
E	fullword boundary
D	doubleword boundary
L	doubleword boundary
A	fullword boundary
Y	halfword boundary
S	halfword boundary
V	fullword boundary
Q	fullword boundary

Table 4-12 Alignment of memory areas in the DS instruction

Programming notes

1. Forcing alignment:
   A DS instruction can be used to align memory areas or constants that would normally not be aligned, on a halfword, fullword or doubleword boundary. To this end, the relevant DC or DS instruction must be preceded by a DS instruction with a duplication factor of zero, and the appropriate type specification (e.g. H, F or D).
2. Redefinition of memory areas:
   A DS instruction with a duplication factor of zero assigns a name and a length attribute to a memory area. Subsequent DS or DC instructions can redefine this area, i.e. fields or constants within it are defined, and accessed individually (see examples (02)).

Fields which are not to be defined in such an area must be skipped using a DS instruction with length specification or with an ORG instruction.

3. Length calculation:
   For memory areas with an implied length of 1 byte (see Table 4-8), the length attribute can be calculated by the assembler if the appropriate values are specified in the chrcon or datcon fields. Here, no length may be specified. The assembler calculates the required length using the values.
4. The DS instruction reserves a memory area, but does not fill it with zeros. The contents of the reserved area cannot be specified in advance.

Examples

```
Name     | Operation | Operands
---------+-----------+-----------------
FIELD1   | DS        | CL80              1 field, length attrib. 80
FIELD2   | DS        | 4CL20             4 fields, lgth.attr.of 20 each
FIELD3   | DS        | 4C                4 fields, lgth.attr. of 1 each
FIELD4   | DS        | F                 1 field, aligned on word
         | .         |                     boundary, length attr. 4
         | .         |
         | DS        | 0F                ┐
AREA     | DS        | XL20              ┘ (01)
         | .         |
         | .         |
RDAREA   | DS        | 0CL50             ┐
         | DS        | CL4               │
PERSNUM  | DS        | CL6               │
NAME     | DS        | CL20              │
         | DS        | CL4               ┘ (02)
DATE     | DS        | 0CL6              │
DAY      | DS        | CL2               │
MON      | DS        | CL2               │
YEA      | DS        | CL2               ┘
         | ORG       | RDAREA+L'RDAREA     (03)
```

(01) The preceding DS instruction with duplication factor zero and type specification F aligns the area AREA on a fullword boundary.

(02) No memory area is reserved for the RDAREA area, but it receives a length attribute.
The subsequent instructions redefine the area, whereby the field DATE is further subdivided. Both the individual fields and the area as a whole may be accessed.

(03) The ORG instruction at the end of the instruction string increments the location counter to the end of the RDAREA area, thus reserving the entire area.

>>>>> See also DC instruction

DXD Define external dummy register

Function

The DXD instruction defines a dummy register.

Format

Name	Operation	Operands
name	DXD	[dup]type[Ln]['val']

name Name
dup Duplication factor
decimal self-defining term or positive absolute parenthesized expression
type Type of dummy register, a single letter
Ln Length modifier
n is a decimal self-defining term or a positive absolute parenthesized expression,
maximum value: dup*n ≤ 4095
val Value specification for the dummy register

Note

The entire format of an operand in the DC instruction may be specified in the operand of the DXD instruction. These extra specifications have no effect on the dummy register itself.

Description

name is the name of the dummy register. The value of the name is the low-order address in the dummy register. This value is only entered during loading. Until then, the name has a provisional value of zero.

dup specifies how often the area specified in the operand is to be reserved. If an expression is specified for dup, the names used in this expression need not have been defined previously.

A **duplication factor with the value zero** is allowed, and has the following effect: a dummy register with a length of zero is defined, and aligned according to its type.

type — If no length modifier is specified, the type determines the alignment of the dummy register, and its length.

Possible type entries:

C	character areas
B	binary areas
P, Z	decimal areas
X	hexadecimal areas
F, H	fixed-point areas
E, D, L	floating-point areas
A, Y, S, V, Q	address areas

Ln — specifies the length of the dummy register, and overwrites the implied length.
If an expression is specified for n, the names used need not have been defined previously.

If a length modifier is specified, the dummy register is not aligned.
For possible values of the length modifier and implied lengths, see DS instruction.

val — Value which may be specified in the operand.

If no length modifier is specified, the assembler uses the value specified to calculate the required length of the dummy register.

The value specified must be valid for the type of dummy register (see description of types of constant).

Programming notes

1. The linkage editor determines the sequence in which all the dummy registers of all the modules to be linked are combined into a dummy register vector.
2. If a dummy register is defined in several assembly units with different lengths and alignments, the linkage editor uses the strongest attribute as the final valid attribute.
3. Dummy registers may be defined at any desired position in the program. Definitions of all the dummy registers need not follow in succession.
4. The DLL currently does not support the use of dummy registers, i.e. such programs must be linked with TSOSLNK, and no dynamic loading mechanism may be used (see "Dynamic Binder Loader / Starter" User Guide [8]).

Example see Appendix A.4

>>>>> See also DC, DSECT and CXD instructions

DROP Drop base address register

Function

The effect of the DROP instruction is to release a hitherto assigned base address register for general utilization.

Format

Name	Operation	Operands
[.sym]	DROP	[reg[,...]]

.sym Sequence symbol

reg General-purpose register; positive absolute expressions, either
- names allocated an absolute value of 0 to 15 or
- decimal self-defining terms of 0 to 15

Description

"reg" denotes the base address register that was assigned in a USING instruction and is now released again. In other words, it is no longer available as a base register.

DROP with a blank operand field releases all base address registers.

Programming notes

1. A register which was blocked with a DROP instruction can be made available at a later stage for addressing via a new USING instruction.
2. No DROP instruction is necessary if the base address is to be altered with a USING instruction.

>>>>> See also USING instruction

DSECT Define dummy section

Function

The DSECT instruction indicates the beginning or continuation of a dummy section.

Format

Name	Operation	Operands
name	DSECT	

name Name

Description

"name" indicates the name of the dummy section.

An additional DSECT instruction with the same name denotes the continuation of the dummy section.

The length attribute of name or &par is 1.

The DSECT instruction, which denotes the beginning of the dummy section, is used to create a new location counter, and is set to an initial value of zero.

No memory area is reserved as a result of defining a dummy section.

Programming notes

1. Those fields to be projected onto a main memory area via the dummy section are described after the DSECT instruction.
2. The names defined in a dummy section may be used as operands in machine instructions. For this, the following is necessary (see example):
   – A base address register must be made available to the assembler via a USING instruction; this register must be effective from the start address of the dummy section.
   – It must be ensured that a base address register is loaded at program execution with the actual address of the memory area on which the dummy section is to be projected.
3. A name which is defined in a dummy section may only be used in a A-type address constant if it is paired with another of an opposite sign in the same dummy section.
4. A dummy register can be defined with a DSECT instruction. The name of the DSECT instruction must then appear as an operand of a Q-type address constant.

Examples

The following example shows two separately assembled programs, PROG1 and PROG2. PROG1 is to read in a record; PROG2 is to process parts of this record. Data is moved from PROG1 to PROG2 via a memory area defined with a DSECT. In this example, the base address register remains the same, and the memory contents are altered. An input is thus read and processed using an overlayed DSECT structure.

```
Name        Operation   Operands
PROG1       START
R0          EQU         0
R2          EQU         2
R3          EQU         3
R4          EQU         4
R14         EQU         14
R15         EQU         15
            .
            .
            BALR        R3,0
            USING       *,R3
            .
            .
READNEXT    RDATA       IN,ERR
            .
            .
            LA          R4,SEN                  (01)
            L           R15,=V(PROG2)
            BALR        R14,R15
            .
            .
            B           READNEXT
ERR         TERM
            .
IN          DS          0CL84
            DS          CL4
SEN         DS          CL80
            .
            .
            END
            .
            .
            .
PROG2       START
            .
            .
            BALR        R2,R0
            USING       *,R2
            USING       BEG,R4                  (02)
            .
            .
BEG         DSECT                               (03)
NR          DS          CL2
NAME        DS          CL2                    }
STREET      DS          CL18                   } (04)
            .                                  }
            .
            END
```

(01) The start address of SEN is loaded into register 4.
(02) Register 4 is assigned as base address register for the dummy section, and the start address of the dummy section is specified.
(03) Definition of the dummy section.
(04) "Template" which is overlayed on the SEN area.

In the second example, a table is to be read. A prerequisite for this is that the table has been filled beforehand, and that each table element has a fixed record format.
The dummy section corresponds to a table element. The table is read by incrementing the base address register by the element length until the end of the table. In this example, the base address register is altered in order to address other memory areas.

```
Name      | Operation | Operands
----------+-----------+-------------------
          | CSECT     |
R1        | EQU       | 1
R2        | EQU       | 2
R15       | EQU       | 15
          | .         |
          | .         |
          | EXTRN     | TABEND
          | EXTRN     | TAB
          | USING     | *,R15
          | USING     | TABELE,R1
          | LA        | R1,TAB
          | LA        | R2,TABEND
LOOP      | MVC       | OUT,STREET
          | .         |
          | .         |
          | LA        | R1,LTABELE(0,R1)
          | CR        | R1,R2
          | BL        | LOOP
          | .         |
          | .         |
TABELE    | DSECT     |
NR        | DS        | CL2
NAME      | DS        | CL2
STREET    | DS        | CL18
LTABELE   | EQU       | *-TABELE
          | .         |
          | .         |
```

>>>>> See also CSECT and XDSEC instruction

EJECT Page feed

Function

The EJECT instruction causes a page feed on the assembler listing.

Format

Name	Operation	Operands
[.sym]	EJECT	

.sym Sequence symbol

Description

The instruction statement after the EJECT instruction in the program appears in the assembler listing on a new page. If such an instruction is already at the start of a new page, the EJECT instruction is ignored.

2 EJECT instructions create a blank page, etc.

>>>>> See also SPACE, TITLE and PRINT instructions

END End assembly

Function

The END instruction denotes the end of an assembly unit and may specify the address of the first instruction to be executed in the program.

Format

Name	Operation	Operands
[.sym]	END	[expr]

.sym Sequence symbol
expr Positive relocatable expression

Description

The operand expr in the END instruction specifies the address of the instruction with which execution is to begin after program loading.

Programming notes

1. The END instruction must always be the last instruction statement in a program.
2. expr may designate an address in a separately assembled program. In this case, expr must be a V-type address constant, or be identified in an EXTRN instruction (see second example).
3. If an END instruction is generated by a macro, the remainder of the source program is skipped and so is not assembled (which also means this cannot be used as a multiple assembly mechanism).

Examples

```
Name      | Operation | Operands
----------|-----------|-----------------
PROG      | CSECT     |
R0        | EQU       | 0
R3        | EQU       | 3
          | .         |
          | .         |
BEGIN     | BALR      | R3,R0
          | USING     | *,R3
          | .         |
          | .         |
          | .         |
          | END       | BEGIN
```

The following example shows an END instruction with an external address which is designated a V-type constant.

```
PROG      | CSECT     |
          | .         |
          | .         |
          | DC        | V(STARTMO)
          | .         |
          | .         |
          | END       | STARTMO
```

>>>>> See also START, CSECT, DSECT and XDSEC instructions

ENTRY Identify entry-point symbol

Function

The ENTRY instruction identifies a symbol that is defined in one assembly unit and is to be referenced from another.

Format

Name	Operation	Operands
[.sym]	ENTRY	name[,...]

.sym Sequence symbol
name Name

Description

"name" denotes a symbol which represents an entry point. In other words, another assembly unit may use this address as a branch destination or data address.

"name" has not yet been defined using the ENTRY instruction. The length attribute of "name" is therefore specified by the instruction statement which defines the name.

Programming notes

1. The name of a control section does not have to be identified by an ENTRY instruction in order to enable refererence from another program.
2. An ENTRY instruction must not contain any name that has been defined in a dummy section.

Example see EXTRN instruction

>>>>> See also EXTRN and WXTRN instructions

EXTRN Identify external symbol

Function

The EXTRN instruction denotes a symbol which is referenced in one assembly unit and defined in another.

Format

Name	Operation	Operands
[.sym]	EXTRN	name[,...]

.sym Sequence symbol
name Name

Description

"name" identifies a symbol which is defined in another assembly unit as an entry point.

As a result of the EXTRN instruction, information regarding the EXTRN reference for the linkage editor is issued.

"name" is defined by the EXTRN instruction for this assembly unit. The assembler assigns a length attribute of 1 and a value of 0.

Programming notes

1. Names defined in an EXTRN instruction may not appear as names of instruction statements in the same program.
2. If names defined as EXTRN are used in arithmetic expressions, they may not appear paired.

Examples of EXTRN and ENTRY

The following example shows how two independent assembly units are linked using the ENTRY and EXTRN instructions. In PROG1, the address IN is identified as external. PROG2 contains IN as a branch destination.

```
Name     | Operation | Operands
---------+-----------+-----------
PROG1    | START     |
R0       | EQU       | 0
R3       | EQU       | 3
R4       | EQU       | 4
R10      | EQU       | 10
R15      | EQU       | 15
         | .         |
         | .         |
         | EXTRN     | IN
         | BALR      | R10,R0
         | USING     | *,R10
         | .         |
         | .         |
         | L         | R15,=A(IN)
         | BALR      | R14,R15
         | .         |
         | .         |
         | END       | PROG1
         | .         |
         | .         |
         | .         |
PROG2    | START     |
R0       | EQU       | 0
R3       | EQU       | 3
R4       | EQU       | 4
R15      | EQU       | 15
         | ENTRY     | IN
         | .         |
         | .         |
IN       | EQU       | *
         | USING     | *,R15
         | .         |
         | .         |
         | BR        | R14
         | END       | PROG2
```

>>>>> See also ENTRY and WXTRN instructions

EQU Equate

Function

The EQU instruction assigns the value and attributes of the expression in the operand entry to a name.

Format

Name	Operation	Operands
name	EQU	expr[,[len][,type]]

name Name
expr Absolute or relocatable expression
len Positive absolute expression; value range: 0 to 2^{24}-1
type Self-defining term, maximum 1 byte long

Description

name The field so designated is assigned the value of expr. The value of the name is relocatable or absolute, depending on whether expr has a relocatable or absolute value.

"name" receives the same length attribute as expr.

If expr is an arithmetic expression, "name" receives the length and type attributes of the name on the extreme left in the operand. If the expression in the operand field consists of an asterisk (*) or of a self-defining term, the length attribute of "name" is 1, and the type attribute is U.

expr Names used in expr need not have been defined previously.

len If len is specified, the name of this value is assigned as the length attribute.

If the length attribute is to be evaluated in macro language, len may only be a self-defining term. If an expression or symbolic parameter is specified for len, a default of 1 is assigned during macro resolution.

type If type is not a self-defining term, the name has the type attribute U (undefined). The type attribute may only be evaluated in macro processing.
The specification may be a self-defining, decimal, hexadecimal, binary or character value.

Programming notes

1. The EQU instruction is frequently used as an aid in structured programming, e.g. in
   - assigning names for register numbers,
   - separating branch addresses and instructions by assigning the branch destination to the current location counter,
   - assigning names for frequently used expressions.

   Examples

```
Name      | Operation | Operands                  
----------|-----------|---------------------------
REG1      | EQU       | 1                      (01)
MARK      | EQU       | *                      (02)
TERM1     | EQU       | A-B*C/2                 ┐
TERM2     | EQU       | B'101011111'            } (03)
TERM3     | EQU       | -10                     ┘
```

   (01) The name REG1 is assigned the value 1. REG1 may then be used as a register name.
   (02) The name MARK is assigned the value of the current location counter.
   (03) The names TERM1, TERM2 and TERM3 are assigned the value of the expresssion in the operand.

2. Expressions whose value is still unknown during program preparation can also be assigned names. The names are assigned the value of the expression in the operand, calculated by the assembler.

3. In macro resolution by the assembler, the EQU instruction is not yet effective.

   Example

```
Name      | Operation | Operands
----------|-----------|---------
REG1      | EQU       | 1
          | MAC       | REG1
```

   This sequence of EQU and macro instructions results in generation of the macro MAC, with character string REG1 as the operand and not with register 1.

4. The explicit length specification in the EQU instruction is frequently used so that a larger area can be addressed when accessing.

   Examples

```
Name      | Operation | Operands
----------+-----------+-------------------------
LOOP1     | EQU       | *                    (01)
LOOP2     | EQU       | *,4                  (02)
LOOP3     | EQU       | *,L'LOOP2            (03)
```

   (01) LOOP1 has the value of the current location counter and the length attribute 1.

   (02), (03)
   In LOOP2 and LOOP3 the length attribute is set to 4 bytes.

5. The explicit type specification alters the original type attribute.

   Example

```
Name      | Operation | Operands
----------+-----------+-------------------------
TERM1     | EQU       | *,,C'A'          } Both names are assigned
TERM2     | EQU       | *,,X'C1'         } type A (address constant)
```

ICTL Input format control

Function

The ICTL instruction can be used to change the default for the begin, end, and continue columns of source program instructions.

Format

Name	Operation	Operands
	ICTL	a[,e[,f]]

a Decimal self-defining term of 1 to 40
e Decimal self-defining term of 41 to 80
f Decimal self-defining term of 1 to 40

Description

a defines the begin column of source program instructions.

e defines the end column of the source program instruction. If not specified, the default applies.

A value of 80 for e means that there are no continue lines.

f defines the continue column for source program instructions. f must be greater than/equal to a.

If f is omitted, this means that there are no continue lines.

Programming notes

1. The defaults of the begin, end, and continue columns are columns 1, 71 and 16 (see section 2.2, "Assembler instruction statements").
2. The ICTL instruction may appear only once in an assembly unit, and must be the first instruction in the program.
3. The ICTL instruction has no effect on macros read in.
4. The operation code of the ICTL instruction may not be generated using variable symbols.
5. Instead of an ICTL instruction, the input format may also be controlled via an SDF option (see "ASSEMBH User Guide" [1]).

LTORG Define literal pool

Function

The LTORG instruction defines a literal pool, and defines its position.

Format

Name	Operation	Operands
[{name / .sym}]	LTORG	

name Name
.sym Sequence symbol

Description

The LTORG instruction causes a literal pool to be created as of the next doubleword boundary.

All valid literals which have occurred since the previous LTORG instruction or since program start are stored in this literal pool.

Literals are aligned in the literal pool according to their type and length. They are stored in the sequence doubleword, fullword, halfword, byte.

All literals which come after the last LTORG instruction are stored at the end of the first control section.

The value of "name" or ".sym" is the address of the first byte of the literal pool. "name" receives a length attribute of 1.

Programming notes

1. If a program contains no LTORG instruction, all literals in a literal pool are stored at the end of the first control section. In that case, the first control section must always be addressable.

   The literal pool is then listed in the assembler listing after the END instruction.

2. An LTORG instruction at the end of each control section ensures that all literals in the respective section are always addressable.

3. Literals which occur more than once in the area of an LTORG instruction are stored only once. An exception here are literals which contain a reference to the location counter. These literals are stored singly together with the current value of the location counter.

OPSYN Redefine mnemonic operation code

Function

The OPSYN instruction assigns the attributes of the mnemonic operation code or macro name in the operand to a name, or ensures that it loses these attributes.

Format

Name	Operation	Operands
name	OPSYN	[code]

name Name or
Mnemonic operation code of an assembler instruction, a machine instruction or a macro instruction or
Macro name

code Mnemonic operation code of an assembler instruction, a machine instruction or a macro instruction,
Macro name or
Name

Description

The name entry is assigned the attributes of the entry in the operand field. Operand entry attributes remain unchanged.

A blank operand field causes "name" to lose the attributes of a mnemonic operation code (see example (02)/(03)).

The same entry in the name and operand fields causes "name" to assume its original attributes again. An identical entry in the name and operand fields is only permitted for mnemonic operation codes of assembler and machine instructions.

Programming notes

1. The OPSYN instruction may be at any point in the source program text, even in macros.
2. The last OPSYN instruction executed remains valid until the next OPSYN instruction. An OPSYN instruction generated in a macro is valid not only for the macro itself, but for all successive instruction statements. If the macro or OPSYN instruction is skipped and not executed, the OPSYN instruction does not come into force.

3. If "name" is the mnemonic operation code of a machine instruction or assembler instruction, this operation code is redefined. If the operand field is blank, this operation code is no longer recognized as a machine or assembler instruction, but is treated as a macro instruction instead.
4. When processing an operation code which is in a macro definition, please note the following:
   - In library macros, all macro statements, including the COPY statement and REPRO instruction, are processed according to the OPSYN instruction which was valid at the first call of the macro.
   - For a macro definition in the source program, all macro statements, including the COPY and the REPRO instructions, are processed according to the OPSYN instruction that was valid at the time the macro definition was read.
   - All remaining instruction statements in library macros and in source program macros are processed according to the OPSYN instruction executed directly before the instruction statement in question.

Example

```
Name     | Operation | Operands
---------|-----------|-----------------
EOP      | OPSYN     | TERM              (01)
         | .         |
         | .         |
         | EOP       |
         | .         |
         | .         |
STORE    | OPSYN     | STH               (02)
STH      | OPSYN     |                   (03)
ABC      | OPSYN     | STORE             (04)
         | .         |
         | .         |
STORE    | OPSYN     |                   (05)
STH      | OPSYN     | STH               (06)
```

(01) The character string EOP is assigned the attributes of the TERM macro. EOP in the operation field on a following line signifies the call of the TERM macro.
(02) Redefines the machine instruction STH; STORE is thereafter interpreted as STH.
(03) With immediate effect, STH is no longer recognized as a machine instruction.
(04) ABC is also assigned, and has the same effect, as STORE.
(05) STORE loses its attributes as a machine instruction with immediate effect, but ABC still defines the machine instruction STH.
(06) Permits the machine instruction STH to assume its original meaning.

ORG Set location counter

Function

The ORG instruction enables the current value of the location counter to be altered.

Format

Name	Operation	Operands
[{name / .sym}]	ORG	[expr]

name Name
.sym Sequence symbol
expr Relocatable expression

Description

"expr" denotes the value to which the location counter is to be set. The ORG instruction without an operand sets the location counter to the next memory location after the highest one addressed thus far in the control section.

Names used in expr must be defined beforehand.

The value of expr must be a relocatable address in the actual control section, i.e. expr may not contain any name defined in another control section.

The end value of the evaluation of expr must not exceed a maximum of 2^{24}-1.

Programming notes

1. expr may not assume any value which precedes the start of the control section containing the ORG instruction.
2. If preceding ORG instructions have reset the location counter, an ORG instruction without an operand may be used to reset the location counter to the memory location that follows the highest one addressed thus far.
3. When resetting the location counter, no EXTRN references, no CXD instructions, and no constants of type A, Q, V, or Y may be overwritten (i.e. no instruction statements which would result in storage of RLD information).

Example

The example below shows the use of an ORG instruction in defining a memory area. The ORG instruction at the end of the instruction sequence increments the location counter to the end of the RDAREA.

```
Name      | Operation | Operands
----------+-----------+---------
RDAREA    | DS        | CL30
          | ORG       | RDAREA
          | DS        | CL4
NAME      | DS        | CL10
          | DS        | CL4
PERSN     | DS        | CL10
          | .         |
          | .         |
          | ORG       |
```

PRINT Print optional data

Function

The PRINT instruction defines those parts of the assembler listing which are to be printed.

Format

Name	Operation	Operands
[.sym]	PRINT	[{BASE / NOBASE}] [{CODE / NOCODE}] [{COPY / NOCOPY}] {[{DATA / NODATA}]} [{GEN / NOGEN}] [{ON / OFF}] [{SINGLE / DOUBLE}]

.sym Sequence symbol

Operands 1 to 7 operands in any sequence; if more than one operand is specified, they must be separated by commas.

Description

BASE After each USING or DROP instruction, the addressable area of the registers (assigned as base registers with USING) is printed out. In addition, 19 characters of the remarks entry from the USING instruction are printed.

NOBASE The addressable areas for the base registers are not printed.

CODE	With regard to instruction statements generated by macros, the effect of PRINT NOGEN is restricted: the text of the source line is suppressed, but the generated code is printed out.
NOCODE	The full effect of PRINT NOGEN is retained.
COPY	Copied instructions are listed.
NOCOPY	Copied instructions are not listed.
DATA	Constants are printed out in full in the assembler listing.
NODATA	Only the first 8 bytes of constants are printed in the assembler listing.
GEN	All instruction statements generated by macro instructions are listed.
NOGEN	Instructions statements generated by macro calls are not listed. Messages generated by MNOTE instructions are, however, printed.
ON	The listing is printed out from this point. PRINT ON is the last instruction that is not listed.
OFF	The listing is not printed out from here on. PRINT OFF is the last instruction that is listed.
SINGLE	Single-line spacing in assembler listing.
DOUBLE	Double-line spacing in assembler listing

Programming notes

A program may contain any number of PRINT instructions. The conditions set by a PRINT instruction remain valid until they are changed by another PRINT instruction. Defaults apply until the first PRINT instruction.

>>>>> See also SPACE, EJECT, TITLE, STACK and UNSTCK instructions

PUNCH Copy text into object module

Function

The PUNCH instruction outputs the characters specified in the operand unprocessed to the object module.

Format

Name	Operation	Operands
[.sym]	PUNCH	'text'

.sym Sequence symbol
text 1 to 80 characters

Description

The assembler stores data shown in the text unprocessed in the object module.

The first character in the text is in the first column of the output format.

Neither a consecutive number nor an identification is output in the object module.

Programming notes

1. For every single quote and ampersand which is to appear in the object module, two corresponding characters must be specified in the text. The two characters are always counted as one.
2. PUNCH instructions may appear at any point in the program or macro definition.
3. The PUNCH instruction may make the object module unusable.
4. The functionality of the PUNCH instruction is no longer supported for module output in LLM format. This is indicated by an error message of error weight 'warning'.

>>>>> See also REPRO instruction

REPRO Copy continuation line into object module

Function

The REPRO instruction outputs the next source program line unprocessed to the object module.

Format

Name	Operation	Operands
[.sym]	REPRO	

.sym Sequence symbol

Description

The assembler stores the source program line which follows the REPRO instruction unaltered into the object module.

The first character on this line appears in the first column of the output format.

Neither a consecutive number nor an identification is output to the object module.

Each REPRO instruction creates a line in the object module.

Programming notes

1. The operation code of the REPRO instruction may not be generated using variable symbols.
2. The REPRO instruction may make the object module unusable.
3. The functionality of the REPRO instruction is no longer supported for module output in LLM format. This is indicated by an error message of error weight 'warning'.

>>>>> See also PUNCH instruction

RMODE Assign load attribute

Function

The RMODE instruction assigns a load attribute to a control section.

Format

Name	Operation	Operands
[{name / .sym}]	RMODE	{24 / ANY}

name Name
.sym Sequence symbol

Description

name refers to a control section of the same name and must correspond to the name of a START, CSECT or COM instruction.

If the name field is blank, the RMODE instruction refers to an unnamed control section.

.sym A sequence symbol means the same as a blank name field.

24 The load attribute 24 has been assigned to the control section, i.e. this section must only be loaded below 16 MB.

ANY The load attribute of the control section may be 24 or 31. In other words, this section can be loaded below and above 16 MB.

The ’Load Attribute’ is described in the manual "Introductory Guide to XS Programming" [7].

If a load module contains several sections with different load attributes, the linkage editor/loader system selects a common load attribute for the entire load module (see "Linkage Editor and Loaders" reference manual [8]).

Information regarding the load attribute of a control section is moved to the ESD record of the object module.

Programming notes

1. The RMODE instruction may appear at any position in the source program. The source program may contain any number of RMODE instructions, but a specified name may appear once only.
2. No RMODE instruction may be set for an unnamed common section (see COM instruction).
3. If no special functions are to be effected, the attributes AMODE = ANY and RMODE = ANY must be assigned to a control section.

Combinations of AMODE and RMODE

If the AMODE instruction is set for a control section, the following combinations are possible with the RMODE instructions for the same control section:

AMODE 24 with RMODE 24
AMODE 31 with RMODE 24 or RMODE ANY
AMODE ANY with RMODE 24 or RMODE ANY

Defaults

The following defaults apply if AMODE and/or RMODE have not been set:

Specified	Default
Neither AMODE nor RMODE	AMODE 24 and RMODE 24
AMODE 24	RMODE 24
AMODE 31	RMODE 24
AMODE ANY	RMODE 24
RMODE 24	AMODE 24
RMODE ANY	AMODE 31

Table 4-13 Defaults for AMODE and RMODE

>>>>> See also AMODE instruction

SPACE Line feed

Function

Using the SPACE instruction, blank lines can be inserted into the assembler listing.

Format

Name	Operation	Operands
[.sym]	SPACE	[no]

.sym Sequence symbol
no Decimal self-defining term

Description

"no" specifies the number of blank lines which are to be printed after the SPACE instruction in the assembler listing. A blank operand entry causes an advance of one line.

If "no" is greater than the number of lines remaining on this page, the SPACE instruction produces a page feed.

>>>>> See also EJECT, PRINT and TITLE instructions

STACK Save USING or PRINT status

Function

The STACK instruction saves the current USING status (i.e. the base registers and the pertinent address range) and/or the current status of the PRINT parameter.

Format

Name	Operation	Operands
[.sym]	STACK	{PRINT[,...][,USING[,...]]} {USING[,...][,PRINT[,...]]}

.sym Sequence symbol

Programming notes

1. The STACK instruction does not alter the current USING status or current PRINT parameter.
2. The PRINT parameter or USING status is stored in a "last-in first-out procedure". In other words, the last information stored is recalled first by the first appropriate UNSTK instruction.
3. In STACK instructions, the operands PRINT or USING may appear up to four times in succession before the first UNSTK instruction must appear with the operands PRINT or USING.

Example

The following example shows a possible application of the STACK USING instruction. The STACK instruction at the beginning of the subroutine enables register 3 to be used differently within the subroutine, without a DROP instruction being required beforehand. UNSTK at the end of the subroutine resets the USING status to the old value, i.e. register 3 is again recognized as base register, with register 5 unknown as in the previous status.

```
Name      | Operation | Operands
----------|-----------|-----------
PROG1     | START     |
R0        | EQU       | 0
R3        | EQU       | 3
R4        | EQU       | 4
R5        | EQU       | 5
          | .         |
          | .         |
          | BALR      | R3,0
          | USING     | *,R3
          | .         |
          | .         |
          | BAL       | R4,PROG2
          | .         |
          | .         |
          | TERM      |
          | .         |
          | .         |
PROG2     | EQU       | *
          | ST        | R3,SAFE
          | STACK     | USING
          | BALR      | R5,0
          | USING     | *,R5
          | L         | R3,NUM
          | AR        | R3,VAL
          | ST        | R3,NUM2
          | L         | R3,SAFE
          | UNSTK     | USING
          | BR        | R4
          | .         |
          | .         |
NUM       | DC        | F'1234'
VAL       | DC        | F'55'
NUM2      | DS        | F
SAFE      | DS        | F
          | .         |
          | .         |
```

>>>>> See also UNSTK instruction

START Define program start

Function

The START instruction identifies the start of an assembly unit, assigns a name to the program or first control section of a program, and sets the location counter to an initial value.

Format

Name	Operation	Operands
[{name / .sym}]	START	[dec[,type][,...]]

name Name
.sym Sequence symbol
dec Self-defining term, which should be divisible by 8;
max. value: FFFFF8
type Attribute identification for control sections (see section 3.2.1.3, "Control section attributes")

Description

name identifies the name of the program or name of the first control section.

A CSECT instruction of the same name denotes the continuation of the first control section.

A START instruction without a name denotes an unnamed control section.

The length attribute of the name is 1.

dec dec denotes the initial value of the location counter of the program.

If dec is not divisible by 8, the location counter is set to the next doubleword boundary.

If dec is not specified, the assembler sets the initial value of the location counter to zero.

type denotes the attributes which are to apply to the program or first control section (see section 3.3.3).

Programming notes

1. The START instruction may not precede any instruction statement which uses or changes the location counter.
2. The START instruction may be replaced by the CSECT instruction to designate the first control section.
3. All instruction statements between the START instruction and the next CSECT or DSECT instruction belong to the first control section. The first control section may be continued at another point in the program by means of a CSECT instruction with the same name.
4. If one or more attribute identifications are specified, the operand dec may not be omitted.
5. If an assembler source program is to be tested using AID, the START instruction must have a name. No LSD information is stored for assembler source programs with an unnamed first control section (see "AID - Debugging of ASSEMBH Programs" [2]).

>>>>> See also END and CSECT instructions

TITLE Listing heading

Function

The TITLE instruction generates page headings in the assembler listing.

Format

Name	Operation	Operands
[{name / .sym}]	TITLE	'text'

name Name, up to four alphanumeric characters long
.sym Sequence symbol
text 1 to 97 characters

Description

"text" forms the page heading in the assembler listing. Every new TITLE instruction in a program causes a page feed and output of the new heading at the top of the subsequent page.

Programming notes

1. If a page heading contains an ampersand or single quote, they must be represented in the text field by two ampersands or two single quotes. The two characters are always counted as one character, and printed out.
2. Only the first TITLE instruction in an assembly unit may ever have a name.
3. For output to the EAM file (not for output to a library), the name entry is shown as an identification in each record of the object module.

>>>>> See also PRINT, EJECT and SPACE instructions

UNSTK Restore USING or PRINT status

Function

The UNSTK instruction restores the USING status or PRINT parameter, previously saved with STACK.

Format

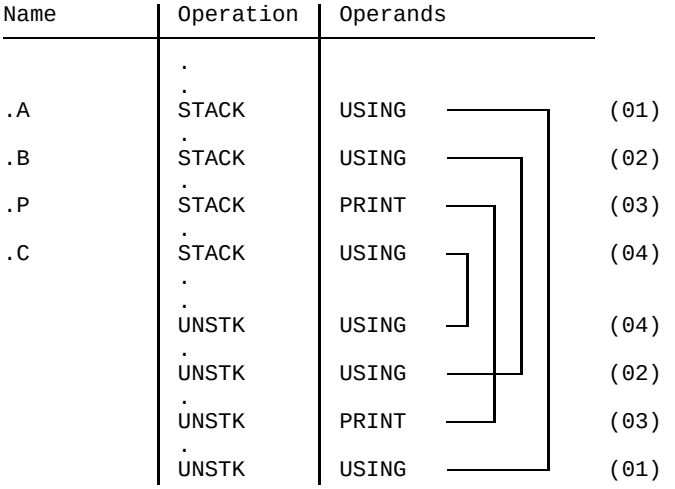
Name	Operation	Operands
[.sym]	UNSTK	{PRINT[,...][,USING[,...]]} {USING[,...][,PRINT[,...]]}

.sym Sequence symbol

Programming note

The UNSTK PRINT or UNSTK USING instruction calls the values that were saved earlier by the last STACK PRINT or STACK USING instruction. The next UNSTK instruction calls the values saved by the corresponding STACK instruction before the last one, and so on.

Example



(01)-(04) The UNSTK instructions restore the current USING or PRINT status of the STACK instruction with the same number.

>>>>> See also STACK instruction

USING Allocate base register

Function

The USING instruction specifies a base address and assigns one or more general-purpose registers to the assembler as base address register(s).

Format 1

Name	Operation	Operands
[.sym]	USING	adr,reg[[,reg][,...]]

.sym Sequence symbol
adr Positive absolute or relocatable expression
reg General-purpose register; positive absolute expressions, either
- names, assigned an absolute value of 0 to 15 or
- decimal self-defining terms of 0 to 15

Description

adr specifies the address value required by the assembler to create displacement addresses.

reg specifies the general-purpose register to be used as a base address register.

Programming notes

1. The USING instruction must be coded prior to the first usage of symbolic names in an instruction statement operand. It is required by the assembler to store addresses in base/displacement form.
2. The USING instruction does not load registers.
   For address accesses to be executed correctly at program runtime, the base registers must also be loaded with base address values. The first is loaded with the BALR or BASR instruction, the others with the L or LM instruction (see Example (02) and "Assembler Instructions" reference manual [3]). The LA instruction is not suitable for this.
3. If the USING instruction is missing, register 0 is used as the base register.
4. Registers 0 and 1 should not be used as base address registers, as they may be altered by various machine instructions and system macros.
5. If an asterisk (*) is used for adr, the USING instruction must follow directly after the BALR instruction, so that BALR and USING denote the same address. If there is a name for adr, this name must contain the address of the instruction at which displacement calculation is to begin (see Examples (01) and (02)).
6. If a base address register is insufficient for an address calculation, other registers must be assigned as base address registers in the USING instruction. These should be loaded with the following base address values (see Example (02)):

   ```
   reg1        adr
   reg2        adr+4096
   reg3        adr+8192
   .             .
   .             .
   ```

7. If the value is altered in a base address register currently in use, and the displacement calculation is continued using this new value, this new value must be assigned with another USING instruction (see Example (03)).
8. If several registers have been assigned as base address registers for one area, the assembler uses the register which provides the smallest displacement address.
   If several registers contain the same value, the register with the highest number is used.
9. If the operand BASE was set in a previous PRINT instruction, the addressable area is printed in the assembler listing after each USING instruction. The 19 character-long remarks entry of the USING instruction is also shown.

Examples

```
Name      Operation  Operands                    
PROG      START
R0        EQU        0                        ┐
R1        EQU        1                        │ valid for all
R3        EQU        3                        } examples
R4        EQU        4                        │ below
R5        EQU        5                        ┘
          .
          .
          USING      BEG1,R3                  ┐
          BALR       R3,0                     } (01)
BEG1      EQU        *                        ┘
          .
          .
          BALR       R3,0                     ┐
          USING      *,R3,R4,R5               │
BEG2      EQU        *                        │
          LM         R4,R5,ACON               } (02)
          .                                   │
          .                                   │
ACON      DC         A(BEG2+4096,BEG2+2*4096) ┘
          .
          .
          USING      BEG2+1000,R3             (03)
          .
          DROP
          .
```

(01) The USING instruction contains the location counter of the instruction with which displacement calculation is to begin.

(02) R3 is to contain the current location counter, so the USING instruction is coded immediately after the BALR instruction.
Using the LM instruction, registers R4 and R5 are loaded with the base address values defined in ACON.

(03) Displacement calculation is continued with the value BEG2+1000 from this USING instruction on.

The following example observes standard linkage rules and allows for the connection of other programming languages (see "ASSEMBH User Guide" [1]).

```
Name        Operation   Operands
            .
            .
            BALR        R10,0
            USING       *,R10
            .
            .
            LA          R15,=A(UPRO2)
            BALR        R14,R15
            .
            DROP        R10
            .
UPRO2       CSECT
            USING       UPRO2,R15               (01)
            .
            .
            BR          R14
            DROP        R15
            .
```

(01) It is not necessary to load register R15 here, as the address has existed since the call to subroutine UPRO2.
A "BALR R15,0" would, however, safeguard against any eventualities.

>>>>> See also DROP instruction

Format 2

Name	Operation	Operands
	USING	*PRV,reg

reg General-purpose register; positive absolute expression, either

- a name, which was assigned an absolute value from 0 to 15, or
- a decimal self-defining term of 0 to 15.

Description

Format 2 of the USING instruction results in the register specified in "reg" being used as the base address register for all instructions concerning dummy registers.

Programming note

The register specified in "reg" must also be loaded with the start address of the memory area for the dummy register vector.

Example of Format 2: see Appendix 11.4

>>>>> See also CXD and DXD instruction

WXTRN Identify conditional EXTRN symbol

Function

The WXTRN instruction identifies a symbol used in one assembly unit, but defined in another. The WXTRN instruction corresponds to the EXTRN instruction, but is processed differently by the linkage editor.

Format

Name	Operation	Operands
[.sym]	WXTRN	name[,...]

.sym Sequence symbol
name Name

Description

The WXTRN instruction suppresses the autolink function of the linkage editor. The linkage editor can only satisfy these external symbols if it can find the appropriate module using another control mechanism.

For other rules, see EXTRN instruction.

>>>>> See also EXTRN and ENTRY instruction

XDSEC Define external dummy section

Function

The XDSEC instruction defines an external dummy section or a reference to an external dummy section.

Format

Name	Operation	Operands
name	XDSEC	[{D[EFINITION] / R[EFERENCE]}]

Description

name denotes the name of an external dummy section.

The length attribute of the name is 1.

DEFINITION, or D
must be specified when defining an external dummy section.

The XDSEC D instruction is used to reserve storage space for the external dummy section.

A new location counter is set up for the definition of an external dummy section, and the initial value set to zero.

The XDSEC D instruction transfers external information to the linkage editor so that is can satisfy accesses to the external dummy section from other programs.

REFERENCE, or R
denotes the reference of an external dummy section to allow accesses to this dummy section.

The instruction XDSEC R sets the location counter to zero, where it remains for the entire section.

The actual address value of an access to an external dummy section is entered by the linkage editor in the appropriate instruction.

Programming notes

1. No EQU instructions may be used in external dummy sections.
2. Arithmetic expressions in operands of an EQU instruction may not differ from XDSEC elements.
3. Type R is assumed for an XDSEC instruction where no operand or an invalid operand is specified. If an external dummy section has already been specified with the same name, its type is taken over.
4. In an assembly unit, an XDSEC instruction may only be type R or type D, not both simultaneously.

Example

In the following example, an input record is initialized in PROG1 and its structure defined. PROG2 and PROG3 process parts of the information from the input record.

```
Name       | Operation | Operands
-----------+-----------+-------------------------
PROG1      | START     |
R0         | EQU       |  0                    ⎫
R1         | EQU       |  1                    ⎪
R2         | EQU       |  2                    ⎬ valid for all 3
R13        | EQU       | 13                    ⎪ programs in the example
R14        | EQU       | 14                    ⎪
R15        | EQU       | 15                    ⎭
           | .         |
           | .         |
           | BALR      | R2,0
           | USING     | *,R2
           | USING     | INXD,R13
           | LA        | R13,IN
           | L         | R15,VPROG2
           | BALR      | R14,R15
           | L         | R15,VPROG3
           | BALR      | R14,R15
           | TERM      |
           | DROP      | R2
*          |           |
IN         | DC        | CL30'ANTON     MUELLER'
VPROG2     | DC        | V(PROG2)
VPROG3     | DC        | V(PROG3)
*          |           |
INXD       | XDSEC     | D                     ⎫
FORENAME   | DS        | CL10                  ⎬ (01)
NAME       | DS        | CL20                  ⎭
           | END       |
           | .         |
           | .         |
           | .         |
PROG2      | CSECT     |
           | .         |
           | .         |
```

```
            USING     *,R15
            USING     INXD,R13
            MVC       OVN,FORENAME          (03)
            BR        R14
            DROP      R15,R13
*
OVN         DS        CL10
*                                          ┐
INXD        XDSEC     R                     } (02)
FORENAME    DS        CL10                  ┘
            END
            .
            .
            .
PROG3       CSECT
            USING     *,R15
            USING     INXD,R13
            MVC       ONAME,NAME            (03)
            BR        R14
            DROP
*
ONAME       DS        CL20
*                                          ┐
INXD        XDSEC     R                     } (02)
NAME        DS        CL20                  ┘
            END
```

(01) Definition of the external dummy section.
(02) External dummy section reference.
(03) Access to the external dummy section.

>>>>> See also CSECT and DSECT instruction

5 Macro language structure

The macro language can be regarded as an additional language, which is also processed by the assembler. The following chapters describe the language elements and instructions which may be used in macro language over and above those in assembler language.

Macro language enables frequently used instruction sequences to be written once only in the form of a macro definition, and to insert it into the program using only one instruction, the macro instruction. Using parameters for control, the inserted instructions may be varied for each macro call. In the macro definition, the insertion of instruction sequences can be made dependent on specific conditions with the aid of conditional assembler instructions. Variable symbols can also be used to create variations in the text of generated instructions and remarks.

5.1 Macro call and definition

The macro call is an instruction in the text of the assembler source program. Macro generation is the result after a macro instruction has been processed. It consists of assembler statements, assembler instructions, macro statements and macro calls, which together carry out the function expected by the macro call.

The operation entry of the macro call denotes the name of the called macro. This name appears in the macro definition in the operation entry of the macro instruction prototype statement (also referred to as prototype statement). The format of the macro instruction operand must correspond to the format of the prototype statement operand (see section 7.1, "Macro call and prototype statement").

The assembler generates macros with the aid of the macro definition. The macro definition is modified in accordance with operand specifications in the macro call. Macro definitions can be written by the user or placed at the user’s disposal as system macros as part of the operating system.

The expanded macros are no longer aligned at positions 10 and 15 in the assembler listing. The 64-character name and operation code fields allow the substituted macro instruction to be mapped as closely as possible on the original instruction in the macro.

The following macro language description refers only to macros written by the user. System macros and their utilization are described in the BS2000 manuals "Executive Macros" [6] and "DMS Introductory Guide and Command Interface" [5].

5.1.1 Storing the macro definition

The definition of a macro can either be included in the source program itself, or stored in a macro library.

If a macro definition in the source program has the same name as a macro definition in a library, the macro definition in the source program is used. If there are macro definitions with the same name in several libraries, the first macro definition found is used (for search hierarchy, see "ASSEMBH User Guide" [1]).

Macro definition in the source program

A macro definition in the source program must always be specified and executed before this macro is called for the first time. If the macro definition is skipped using conditional statements, it is not read in either. The macro is then regarded as unknown in the source program. If such a macro is called, the macro library is searched for the definition, and if none is available there, the macro call is reported as an error (unknown operation code; see "ASSEMBH User Guide" [1]).

If the same name is used for different macro definitions in the source program, the last macro definition loaded is valid until the appearance of the next definition of the same name. In this case, the assembler displays a warning.
Macro definitions skipped by means of conditional assembly statements are also ignored in this instance.

Macro definition within a macro definition

It is also possible for a macro definition to be nested within another (inner macro definition), i.e. to read a macro definition on the basis of a macro generation (see also section 5.1.2, "Format of the macro definition").

This inner macro definition must also be executed before it is called. Only then is it recognized in the outer macro.

Once the inner macro definition has been executed for the first time, it may also be called independently of the outer macro.

The outer macro definition can be in the source program or stored in a library. The inner macro definition is stored as if it were loaded from a library.

The nesting level for inner macro definitions is arbitrary, depending on memory capacity.

Macro definition in a library

A macro definition can be made accessible to many programs if it is stored in a library. To enable the assembler to find the macro definition, the appropriate library must be allocated prior to assembly. (see "ASSEMBH User Guide" [1]).

If a macro definition is included in a library, it is only read in if its macro call is executed. If the call is skipped and not executed, the macro definition is not read in.

5.1.2 Format of the macro definition

A macro definition consists of the following four sections, in the given order:

- **The macro definition header MACRO**
  This denotes the start of a macro definition and must always be the first statement in the definition (see section 7.2, "Description of macro statements").
- **The macro instruction prototype statement**
  This specifies the name of the macro and the symbolic parameters which occur in this macro definition.
  The operand format of the prototype statement defines the operand format of the relevant macro instruction (see section 7.1, "Macro call and prototype statement").
- **No, one or several model statement(s)**
  These produce, after assembly, the required sequence of instructions.
  Model statements may be:
  - assembler instructions, except the ICTL instruction,
  - machine instructions,
  - macro instructions, and
  - macro statements.
- **The macro definition trailer MEND**
  This identifies the end of a macro definition and must always be the last statement in the definition (see section 7.2, MEND).

In addition to the above-mentioned components, there may be remarks lines which can be at any required position in the macro definition (see section 5.2, "Instructions and remarks").

Example

The example shows the possible logic of a macro definition in a simple form.

```
Name        | Operation | Operands                    |
------------+-----------+-----------------------------+
            | MACRO     |                             | (01)
&LABEL      | VARPAR    | &A,&B                       | (02)
            | LCLC      | &C                          | (03)
&LABEL      | MVC       | &A,&B                       | (04)
            | .         |                             |
            | .         |                             |
&C          | SETC      | 'WORK'                      | (05)
            | .         |                             |
            | .         |                             |
            | AIF       | ('&C' EQ 'WORK').ONE        | (06)
            | AIF       | ('&C' EQ '').TWO            | (07)
            | .         |                             |
            | .         |                             |
            | AGO       | .TWO                        | (08)
            | .         |                             |
            | .         |                             |
.ONE        | ANOP      |                             | (09)
            | .         |                             |
            | .         |                             |
.TWO        | ANOP      |                             | (10)
            | .         |                             |
            | .         |                             |
            | MEND      |                             | (11)
```

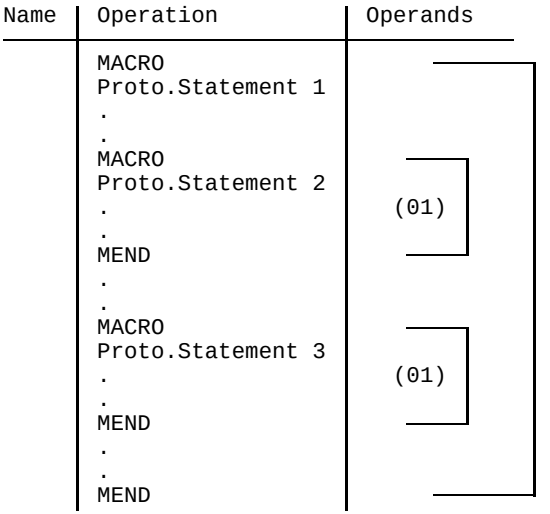
(01) Macro definition header
(02) Prototype statement with the macro name VARPAR and the symbolic parameters &LABEL, &A and &B
(03) Up to (10) model statements:
(03) Definition of the local SET symbolic parameter &C
(04) The symbolic parameter &LABEL represents the symbol of the macro call via replacement in the prototype statement
(05) Value assignment for &C
(06), (07)
Conditional branches, branch destination is dependent on the contents of &C
(08) Unconditional branch
(09), (10)
Definitions of the branch destinations .ONE and .TWO
(11) Macro definition trailer

5.1.3 Inner macro definition

A macro definition within another is designated an inner macro definition.

This inner macro definition cannot be generated with variable symbols, i.e. no text is replaced (see section 5.3.2, "Variable symbols") while the macro definition is being inserted. The outer macro can only be used to control the insertion or non-insertion of the inner macro definition.

A macro definition which contains an inner macro definition can only be created as shown below.



(01) Additional inner macro definitions can be nested here.

5.2 Instructions and remarks

Just as the text of an assembler source program, the text of an assembler macro consists of a series of instructions and remarks. The additional facilities offered by the macro language for instructions and remarks are described below. The details given regarding assembly language structure in chapter 2 are also applicable here.

Instructions

Macro language instructions consist basically of five entries:
- the name entry,
- the operation entry,
- the operand entry,
- the remarks entry, and
- the continuation character.

These entries must appear in the above sequence and, except for the continuation character, must be separated from one another by at least one blank.

The name, operation, and operand entries in various instructions, including assembler and machine instructions, can be generated by means of **variable symbols** in accordance with requirements.

For one macro language instruction, nine continuation lines are allowed. Here, the macro instruction, prototype statement, the GBLx and LCLx instructions, plus Format 2 of AIF and AGO have special significance; there is an alternative format for them (see section 7.1.4, "Alternative statement format").

Remarks

Remarks may appear anywhere in the macro definition. In macro language, as in assembly language, there are two ways of denoting remarks:

* With an asterisk in the begin column.
These remarks lines are copied by the assembler during macro resolution, and printed out in the assembler listing.

.* With a period in the begin column, followed by an asterisk.
These remarks lines are used only to document the macro definition. During macro resolution, these are copied and printed by the assembler only for macro definitions in the source program, not for macro definitions in libraries.

As in the remarks entry of instruction statements, variable symbols are not replaced in remarks lines. Variable remarks lines may only be generated via the instruction MNOTE *,... (see section 7.2).

Deactivating the function of instruction statements

If in the name entry of instruction statements symbolic parameters or SETC symbols are allowed, remarks can be generated from instruction statements in the macro definition or source program, i.e. their function can be deactivated.

In such a case, the symbolic parameter or SETC symbol is assigned an * for macro generation (.* is not allowed). This results in the instruction being interpreted as a remark.

5.3 Name entry

The name entry may contain a sequence of up to 64 letters and digits, which are used to identify an instruction.

The name entry is optional. If it exists, it must begin in the begin column. If the begin column is blank, the assembler assumes there is no name entry and interprets the subsequent characters as the operation code.

In the name entry of an instruction statement, there may be:
- a name (see section 2.3),
- a sequence symbol,
- a variable symbol or
- a concatenation of variable symbols and alphanumeric characters.

5.3.1 Sequence symbols

A sequence symbol in the name entry enables the conditional assembly statements AIF and AGO to refer to this instruction, i.e. to identify it as a branch destination. Thus, the order in which instructions are processed can be varied.

Sequence symbols are local symbols, i.e. they are only recognized in the macro definition in which they were defined. If the same sequence is used in and outside a macro definition, or in two different macro definitions, it is always regarded as a separate symbol.

A sequence symbol in the name entry of a generated instruction is not printed in the assembler listing.

Rules

- Sequence symbols may consist of a maximum of 65 characters.
- The first character of a sequence symbol must be a period (.), the second a letter. These may be followed by up to 63 additional letters and/or digits.
- Sequence symbols in the name entry of instructions may only be written in this standard format.
- Sequence symbols in the operand entry may also be written in generated format. Sequence symbols in generated format consist of a period, followed by a variable symbol or a concatenation of variable symbols and alphanumeric characters (see section 5.5, "Operand entry").
- No space characters are permitted in a sequence symbol.
- A maximum of 2^{15}-1 sequence symbols are allowed per macro.

Examples of valid sequence symbols

```
.LOOP                   .A123456789
.LOOP_2                 .A@B4
.Loop                   .$ABC
                        .ABC&PARAM
```

Examples of invalid sequence symbols

```
AREA                    (first character not a period)
.1ABC                   (second character not a letter)
.LOOP*                  (contains the special char. *)
.LOOP␣1                 (contains a space character)
```

5.3.2 Variable symbols

Variable symbols are used to replace text in the name, operation, and operand entries of an instruction. Their values may be assigned via macro statements or directly by the assembler. Text is replaced by the actual value. There is no replacement of text in the remarks entry. When used in conditional assembly statements, variable symbols can be used to control assembly.

The assembly or macro language instruction statements in which variable symbols are allowed can be seen in the relevant format descriptions. Variable symbols are generally permitted in the name entry in machine instructions.

A maximum of 2^{15}-1 variable symbols are allowed per macro definition.

Rules

- A variable symbol may consist of up to 64 characters.
- The first character must be an ampersand (&).
- Space characters are not allowed in a variable symbol.

Characters which replace a variable symbol in text replacement must adhere to the syntax rules for the entry in which the variable symbol appears.

Example

`&PAR` in the name entry cannot be replaced by `NAME␣BALR`

On the other hand, except for a macro call, two or more operands can be generated from a variable symbol within the operand entry.

Example

`&PAR` in the operand entry can be replaced by `FIELD1,FIELD2`

Variable symbols include:

- **symbolic parameters;**
  These are defined in the prototype statement and are assigned new values by the programmer for each macro instruction.
- **system variable symbols;**
  These begin with the character string &SYS... and have fixed meanings. The assembler assigns them a value when processing a macro definition or at the start of an assembly.
- **SET symbols;**
  These are variable auxiliary fields to which the programmer can assign a value in a macro definition by using macro statements.

Symbolic parameters and some of the system variable symbols are local symbols. In other words, their values are always reset at the start of macro processing. They are therefore only recognized within a macro.

SET symbols may be local or global symbols. If they have been defined as global, they can be used to transfer values between macros or between a macro and the assembler source program.

Note

A detailed description of variable symbols and their application is given in chapter 6, "Variable symbols".

5.3.3 Generated variable symbols

As a rule, the name of a variable symbol can be generated by means of one or more additional variable symbols.

In doing so, the characters which represent the result of the text replacement must adhere to the syntax rules of the entry in which the generated variable symbol is used.

Format of generated variable symbols

&(ele)[(d)]

ele may be:
- variable symbols: &par
- subscripted SET symbols (see section 6.2): &par(d)
- generated variable symbols: &(ele)
- generated subscripted SET symbols: &(ele)(d)
- alphanumeric characters: val
- concatenation of above basic elements (see section 5.2.1.4)

d may be:
- index, arithmetic macro expression: arexp
- operand sublist with arithmetic macro expressions (see section 7.1.2): arexp,arexp[,...]

Examples

```
&(&PAR1.&(&PAR2(D1)).AB)  gives   &A1AB   if   &PAR1      SETC  'A'
                                               &PAR2(D1)  SETC  'B'
                                               &B         SETA  1

&(&(&(AB)(D1))(D2))       gives   &PAR2   if   &AB(D1)    SETC  'PAR1'
                                               &PAR1(D2)  SETC  'PAR2'
```

5.3.4 Concatenation of variable symbols and alphanumeric characters

Variable symbols may be chained to each other or to alphanumeric characters. The character string produced in the generated text as a result of the concatenation must adhere to the syntax rules of the entry in which the concatenation appears.

– If alphanumeric characters are to follow a variable symbol, they must be separated by a period. If the period is to appear in the generated text, two periods must be entered in the macro definition.

  Example (the actual value of `&PARAM` is `NAME`)

`&PARAM.ABC`	gives	`NAMEABC`
`&PARAM..ABC`	gives	`NAME.ABC`

– If a variable symbol is to follow alphanumeric characters, no period is allowed. If a period is to appear in the generated text, only one period may be entered in the macro definition.

  Example

`ABC&PARAM`	gives	`ABCNAME`
`ABC.&PARAM`	gives	`ABC.NAME`

– If variable symbols are to be chained, they can be chained by stringing them together or by separating them with a period. If the period is to appear in the generated text, two periods must be entered here in the macro definition.

  Example (The actual value of `&PARAM1` is `NAME1` and of `&PARAM2 NAME2`)

`&PARAM1&PARAM2`	gives	`NAME1NAME2`
`&PARAM1.&PARAM2`	gives	`NAME1NAME2`
`&PARAM1..&PARAM2`	gives	`NAME1.NAME2`

  In the name entry, a concatenation of variable symbols and alphanumeric characters may only be used in assembler and machine instructions, not in macro instructions. In the case of operation and operand entries, the utilization of such a concatenation is dependent on the format of the instruction.

5.4 Operation entry

The operation entry may contain:
- the mnemonic operation code of a macro statement, an assembler or machine instruction, or
- the name of a system or user macro (macro call), or
- a variable symbol, or
- a concatenation of variable symbols and alphanumeric characters.

An operation entry is mandatory. It must be separated from the name entry by at least one space character. If there is no name entry, it must begin at least one position to the right of the begin column.

A valid operation entry consists of the mnemonic operation code of an assembler instruction, a machine instruction, or macro statement. Names of user macros must be created according to the rules for names (see section 2.2.1, "Name entry").

Variable symbols in the operation entry

The operation entry may be generated using variable symbols or through concatenation of variable symbols and alphanumeric characters. Characters which replace a variable symbol must adhere to the syntax rules of the operation entry.

The following instructions may **not** be generated with variable symbols:
- all macro instruction statements
- the COPY instruction
- the ICTL instruction
- the MNOTE instruction, and
- the REPRO instruction.

All other assembly language instruction statements and also names of system and user macros can be generated using variable symbols.

Variable symbols in the operation entry are subject to the same rules as variable symbols in the name entry (see section 5.3.2 and chapter 6, "Variable symbols").

Example

```
Name        | Operation | Operands
------------------------------------------
* Macro definition
            | MACRO     |
            | EXOP      | &OP,&BASREG
            | .         |
            | .         |
            | &OP       | &BASREG,0
            | USING     | *,&BASREG
            | .         |
            | .         |
            | MEND      |

* Macro call 1
            | EXOP      | BASR,2

* Generated instruction statements
            | BASR      | 2,0
            | USING     | *,2

* Macro call 2
            | EXOP      | BALR,2

* Generated instruction statements
            | BALR      | 2,0
            | USING     | *,2
```

5.5 Operand entry

The operand entry of instruction statements in macro language may consist of one or more operands, which in turn may contain one or more expressions.

Operands must be separated by commas. There may be no space characters, except in the alternative statement format (see 7.1.4), between operands and separating commas.

The operand entry is optional. If specified, it must be separated from the operation entry by at least one space character.

5.5.1 Variable symbols in the operand entry

Variable symbols may be used in the operand entry of assembler instruction statements and macro instruction statements. In assembler instruction statements, they are used to replace text. In macro instruction statements, whether or not a variable symbol is possible, and how it is evaluated, can be seen from the format.

Variable symbols in the operand entry are subject to the same rules as variable symbols in the name entry (see section 5.3.2 and chapter 6, "Variable symbols").

Example

```
Name        | Operation | Operands
------------+-----------+-----------------
* Macro definition
            | MACRO     |
            | EX1       | &PAR1,&PAR2
            | MVC       | &PAR1,&PAR2
            | .         |
            | .         |
            | MEND      |

* Macro call
            | EX1       | FIELD1,FIELD2

* Generated instruction statements
            | MVC       | FIELD1,FIELD2
            | .         |
            | .         |
FIELD1      | DS        | CL4
FIELD2      | DC        | C'ABCD'
```

5.5.2 Sequence symbols in the operand entry

Besides macro expressions, the operand entry of AIF and AGO instructions may contain sequence symbols. The sequence symbols in the operand entry specify the instruction statements to which branches are to be made once the AIF and AGO instructions have been processed.

Sequence symbols may be entered in the operand entry in standard format (see section 5.3.1) or in generated format.

A sequence symbol in standard format begins with a period (.). The second character must be a letter, and this may be followed by a further 63 letters and digits.

A sequence symbol in generated format also begins with a period. This is followed by a variable symbol or a name which is chained with a variable symbol.

Examples of sequence symbols in generated format

```
.&LOOP
.LOOP&NAME
.&Loop
.NAME123&LOOP
.&LOOP.NAME
```

5.5.3 Macro expressions

In macro language, expressions are also the basic components of instruction statement operands. They are composed of elements and operators.
In macro language, there are arithmetic macro expressions, character expressions, relational expressions and Boolean expressions.

The type of expression and its ensuing calculation is determined by the operators. Details of the various expressions and possible elements of expressions are illustrated in Figure 5-1.

Arithmetic macro expressions and Boolean expressions may consist of only one element each, without operators. The assembler interprets these simple expressions accordingly, i.e. as arithmetic or Boolean expressions.

Character expressions consist of one element only, and contain no operators.

Relational expressions, on the other hand, comprise two elements and one operator.

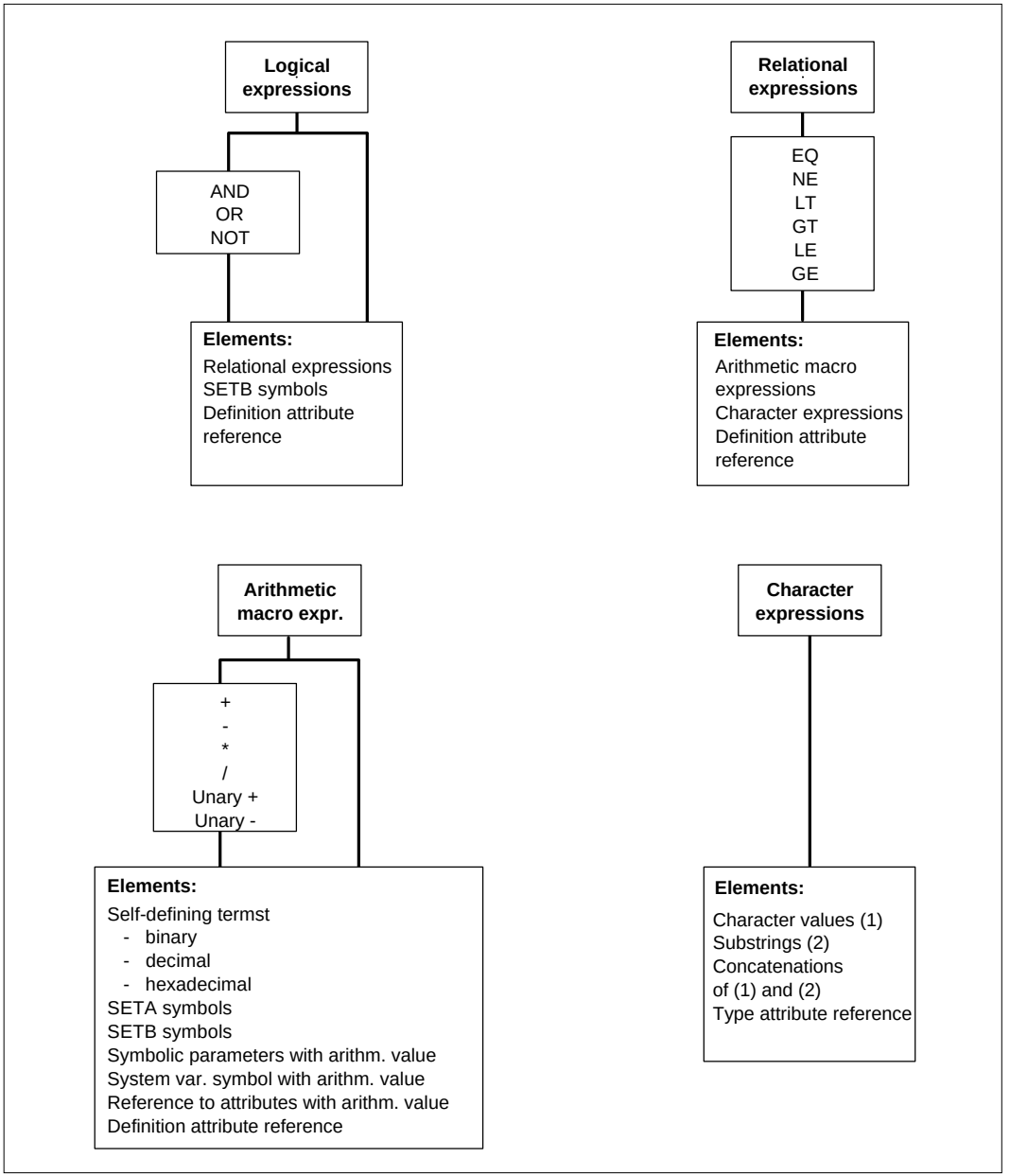


Fig. 5-1 Macro expressions

5.5.4 Character expressions

A character expression may be:
- a character value,
- a character substring,
- a concatenation of character values and character substrings or
- a type attribute reference (see section 5.2.3.5).

A character expression contains no operators.

5.5.4.1 Character value

A character value consists of any combination of characters, enclosed in single quotes.

```
Format of character values    'val'
```

val — Character string or
variable symbol or
concatenation of variable symbols and alphanumeric characters.

A variable symbol in a character value is replaced by its actual value.

If a variable symbol which represents a character value is used as an element in a relational expression, both single quotes may be omitted.

If a SETA or SETB symbol is used, the result is the character representation of the decimal or Boolean value, without a sign (absolute value) and without leading zeros.

A character value may consist of a maximum of 1020 characters.
Similarly, the resolution of a variable symbol may only produce a character string of up to 1020 characters.

Single quote in character values

If, after generation, the resolution of a character value is to contain a single quote, two single quotes must be written in the character value itself.

Examples

```
'NUM''ONE'      gives     NUM'ONE
'L''SYMBOL'     gives     L'SYMBOL
''''            gives     '
```

Ampersand in character values

An ampersand in a character value is interpreted by the assembler as the start of a variable symbol. Unlike the resolution of two single quotes, two ampersands in a character value are stored as two ampersands. In other words, if there are two consecutive ampersands in a character value, then they are replaced by the same value in texts as well.

A single ampersand in the resolution of a character value can only be generated by means of a character substring (see below).

Example

```
Name        | Operation | Operands
------------|-----------|-------------------------
&FIRMA      | SETC      | 'NAME && CO'          (01)
            | DC        | C'&FIRMA'             (02)

* Generated instruction statement
            | DC        | C'NAME && CO'         (03)

*Generated constant  NAME & CO
```

(01) The variable symbol &FIRMA is assigned the value NAME && CO.
(02) The C-type constant is defined with the variable symbol.
(03) After macro resolution, the contents of the C-type constant are NAME && CO. This value adheres to assembler syntax and is assembled as NAME & CO.

5.5.4.2 Character substring

Substrings enable a section to be accessed within a character value.

Format of Character Substrings `[(dup)]'val'(a,b)`

dup — Duplication factor; arithmetic macro expression

The duplication factor is only allowed for substrings in the operand entry of SETC instructions. Substrings in relational expressions may have **no** duplication factor.

val — Character value

a — Number of the character at which the section of the character value is to begin;
decimal self-defining term or arithmetic macro expression

b — Number of characters which the section of the character value is to contain;
decimal self-defining term or arithmetic macro expression

When a character substring is resolved, the section of the character value is established first. The duplication factor is evaluated thereafter.

Following the evaluation of the duplication factor, a substring, like its resolution, may consist of a maximum of 1020 characters only.

In a substring, if value "a" is greater than the number of characters in the character value, the result is a blank character string. If value "b" is greater than the number of characters, only the existing characters are inserted in the character value as the result (see examples).

Examples

```
(3)'TEXT'(2,2)    gives    EXEXEX
'VAL &&'(1,5)     gives    VAL &
(3)'&PAR'(1,&A)   gives    ABABAB  } if &PAR has an actual value of ABC
'&PAR.%4'(1,4)    gives    ABC%    } and &A has the actual value of 2.
'ABCD'(12,4)      gives    empty string
'ABCD'(1,10)      gives    ABCD
```

5.5.4.3 Concatenation of character values and substrings

Character values and substrings may be chained together in any order:

The following applies to all concatenations:

– There **must** be a period between two successive single quotes:

Examples

```
'ABC'.'DEF'              gives   ABCDEF
'ABC'.'DEF'(2,1)         gives   ABCE
```

– There **must** be a period between the single quote and the left parenthesis (duplication factor).

Examples

```
'ABC'.(2)'XY'            gives   ABCXYXY
'BIN'.(2)'XY'(1,1)       gives   BINXX
```

– There **may** be a period between the right parenthesis and the single quote (substring).

Examples

```
'PAR123'(1,3).'AB'       gives   PARAB
'PAR123'(1,3)'AB'        gives   PARAB
```

– There **may** be a period between the left and right parentheses.

Example

```
'PAR123'(1,3).(2)'AB'    gives   PARABAB
'PAR123'(1,3)(2)'AB'     gives   PARABAB
```

Note

Character substrings in relational expressions may have no duplication factor in this case as well.

5.5.5 Arithmetic macro expressions

An arithmetic macro expression is composed of elements and arithmetic operators (see section 2.5.1) or of only one element without operators.

The following elements are permitted in an arithmetic macro expression:

- binary, decimal and hexadecimal self-defining terms (see section 2.5.2),
- SETA symbols (see section 6.2),
- SETB symbols (see section 6.2),
- SETC symbols, symbolic parameters and system variable symbols, provided they have an arithmetic, binary, or hexadecimal value (see 6.1 to 6.3), and
- attribute references with an arithmetic value (for length attribute, count attribute, number attribute, scaling attribute and integer attribute; see section 5.5.8),
- definition attribute references (see section 5.5.8.7).

The elements listed above are described in the relevant sections of this manual.

In an arithmetic macro expression, the sequence of processing can be altered by means of parentheses. Parentheses may be nested.

Rules

- An arithmetic macro expression may not begin with any operator other than unary plus and unary minus.
- In an arithmetic macro expression, one element may not follow immediately after another.
- Unary plus and unary minus may follow immediately after all other operators.
- The final values of the calculation of arithmetic macro expressions must be between -2^{31} and $+2^{31}-1$.

Examples of valid arithmetic macro expressions

```
&AREA+X'2D'
&EXIT-S'&ENTRY+1
&AREA+X'2D'/(&EXIT-S'&ENTRY+1)
I'&N/25
50
```

Calculation of arithmetic macro expressions

1. Each element is assigned its numeric value.
2. Arithmetic operations are performed from left to right. Multiplication and division precede addition and subtraction.
3. In expressions containing parentheses, the values in the parentheses is calculated first. For multiple parentheses, the inner parenthesized expression is calculated first.
4. Division by zero is allowed and returns a result of zero.

Note

If SETC symbols, symbolic parameters, and system variable symbols have no arithmetic value, a flag is generated (not in F-ASSEMB-COMPATIBLE mode), and 0 is used as the substitution value.

5.5.6 Relational expressions

A relational expression is composed of two elements and one relational operator. The result of a relational expression may be 0 or 1 (false or true).

The following relational operators are permitted:

EQ equal
NE not equal
LT less than
GT greater than
LE less than/equal
GE greater than/equal

The elements of the relational expression determine whether an arithmetic relation or character relation is involved.

In an **arithmetic relation**, at least one of the elements must be an arithmetic macro expression.

In a **character relation**, both elements are character expressions.

Rules

- Relational operators must be separated from elements by a space character.
- A relational expression may be parenthesized.
- If variable symbols are used as character values in a relational expression, single quotes, which in all other cases denote a character value, may be omitted.
- In a character relation, elements up to a length of 1020 characters may be compared. If the elements in a character relation are of unequal length, the shorter element is always regarded as less than the longer one.

Examples of valid relational expressions

&CHAR1 and &CHAR2 are to be SETC symbols; &AR1 and &AR2, SETA symbols. &PAR1 is a symbolic parameter to which a character string has been assigned; &PAR2 is a symbolic parameter to which an arithmetic value has been assigned.

`'FIELD' NE '&CHAR1'`	character relation
`'FIELD' NE &CHAR1`	character relation
`&PAR1 EQ &CHAR1`	character relation
`&AR1 GT &AR2`	arithmetic relation
`&AR1 GT 16*&AR2+4`	arithmetic relation
`&AR1 GT 20`	arithmetic relation
`&PAR2 EQ &CHAR1`	character relation
`&PAR2 EQ &AR1`	arithmetic relation

5.5.7 Boolean expressions

Boolean expressions are composed of elements and logical operators or only one element without operators. They may contain as the result only the logical values 0 (false) or 1 (true).

Elements of a Boolean expression may be:

- SETB symbols (see section 6.2),
- relational expressions (see section 5.5.6) and
- definition attribute references (see section 5.5.8.7).

The following logical operators are allowed:

AND	logical AND
OR	inclusive OR
NOT	negation

The sequence of processing in a Boolean expression can be altered by means of parentheses. Parentheses may be nested.

Rules

- Logical operators must be separated from elements by a space character.
- A Boolean expression may not begin with AND or OR.
- In a Boolean expression, one element may not follow immediately after another.
- Each element in a Boolean expression may be parenthesized. In this case, no space character is necessary between operator and parenthesized element.
- The operators AND and OR may not be combined.
  The only combination allowed with NOT is AND NOT, or OR NOT.

Examples of valid Boolean expressions

```
&PAR1 AND &PAR2
&PARA OR &PARB
NOT &B AND &C
NOT(&BIN1 AND &BIN2)
(&BINA AND NOT &BINB)OR(&BINB AND NOT &BINA)
```

The following examples show valid Boolean expressions in which character relations and arithmetic relations are used as elements.

```
&AREA+2 GT 29 OR &AR1
(&AREA+2 GT 20)OR(&AR1)
NOT &AR1 AND &AREA+2 GT 20
NOT &AR1 AND(&AREA+2 GT 20)
```

Calculation of Boolean expressions

Boolean expressions are reduced to a single value according to the following rules:

1. Each element in the Boolean expression is evaluated and assigned the logical value 0 or 1 (false or true).
2. Logical operations are performed from left to right. NOT, however, is processed prior to AND, and AND prior to OR.
3. In expressions containing parentheses, the parentheses are resolved from the inside out.

5.5.8 Attribute references

The assembler assigns attributes to names and variable symbols. These attributes can be referenced, for example, so that the execution of certain instructions can be made dependent on the corresponding attribute.

```
Format of an attribute reference       attr'par
```

attr attribute designation (see below)
par name or
variable symbol

Each attribute has a specific designation with which it is accessed.

Type attribute	T
Length attribute	L
Scaling attribute	S
Integer attribute	I
Count attribute	K
Number attribute	N
Definition attribute	D

The following table shows which attributes can be referenced for names and for different variable symbols.

	Names	Symbolic Parameters	SET Symbols	System Variable Symbols
T	X	X	X	X
L	X	X	X	X
S	X	X*	only &SETC*	only &SYSLIST(n)*
I	X	X	only &SETC*	only &SYSLIST(n)*
K		X	X	X
N		X	X	only &SYSLIST(n) and &SYSLIST
D	X	X*	only &SETC*	only &SYSLIST(n)*

* Only valid if the value is a name

Table 5-1 Attribute references for names and variable symbols

Name attributes
The value of the attribute is evaluated using the data which represents the name.

To do this, the name must be defined, i.e. it must be in the name entry of an assembler instruction or machine instruction or in the operand entry of an EXTRN or WXTRN instruction. The instruction in which the name is defined must be in the assembler source program. It is also possible for the name to be defined at a later point in the source program (see "ASSEMBH User Guide" [1], "Lookahead mechanism"). An exception here is the definition attribute reference, which queries whether a name is already defined at the time of the query.

Variable symbol attributes

The value of the attributes of variable symbols is evaluated using the actual value assigned to the variable symbol.

The attributes of **SET symbols and system variable symbols (except &SYSLIST)** are always calculated from the actual value.

The attributes of **symbolic parameters and the system variable symbol &SYSLIST** are calculated using the operands of the corresponding macro call.

If symbolic parameters or &SYSLIST(n) are assigned new values via a SETC instruction in the macro, the attributes are likewise evaluated on the basis of these current values.

If the operand of the macro call is a macro expression, the corresponding symbolic parameter will be assigned the attributes of a character constant.

If the operand of an inner macro instruction is an operand sublist, either the attributes of the sublist or any individual operand of the sublist may be designated. The type, length, integer, and scaling attribute of a sublist is the same as the corresponding attribute of the first element from the sublist.

5.5.8.1 T' Type attribute reference

The type attribute of a name or variable symbol is a single letter.

The type attribute reference may only be used as an element of character expressions. It must always stand alone.

Examples

```
Name        | Operation | Operands
------------+-----------+-------------------
&A          | SETC      | T'&PAR
&B          | SETB      | (T'&X NE T'&Y)
.C          | AIF       | (T'&PAR NE 'F').D
```

- The following type attributes apply to names and symbolic parameters which denote DC, DS, DXD and CXD instructions. An appropriate name as a value must be assigned to the symbolic parameter in the macro call.

  A A-type address constant, implied length, aligned on a fullword boundary, CXD instruction
  B Binary constant
  C Character constant
  D Floating-point constant, double precision, implied length, aligned on doubleword boundary
  E Floating-point constant, single precision, implied length, aligned
  F Fullword fixed-point constant, implied length, aligned on a fullword boundary
  G Fixed-point constant, explicit length
  H Halfword fixed-point constant, implied length, aligned on a halfword boundary
  K Floating-point constant, explicit length
  L Floating-point constant, extended precision, implied length, aligned
  P Decimal constant, packed
  Q Relative address in an external dummy section
  R A-, S-, V- or Y-type address constant, explicit length
  S S-type address constant, implied length, aligned
  V V-type address constant, implied length, aligned
  X Hexadecimal constant
  Y Y-type address constant, implied length, aligned
  Z Zoned decimal constant

Example

```
Name        | Operation | Operands
------------------------------------------
* Macro definition
            | MACRO     |
            | EXTYP1    | &PAR
            | AIF       | (T'&PAR EQ 'F').FCON
            | .         |
            | .         |
.FCON       | DC        | ...
            | MEND      |

* Program with macro call
CONST       | DC        | F'3'
            | .         |
            | .         |
            | EXTYP1    | CONST
            | .         |
            | .         |
```

The symbolic parameter &PAR is assigned the value CONST.

Type of CONST: C
Type of &PAR: C

- The following type attributes apply for names and symbolic parameters,
  – which denote instruction statements other than DC, DS, DXD and CXD instructions or
  – which are defined in the operand entry of an EXTRN or WXTRN instruction.

  Here too, the symbolic parameter must be assigned a name as a value in the macro instruction.

  I Machine instruction
  J Name of a control section
  M Macro call
  T External name

Example

```
Name        | Operation | Operands
------------------------------------
* Macro definition
            | MACRO     |
            | EXTYP2    | &PAR
            | .         |
            | .         |
&PAR        | MVC       | ...
            | MEND      |

* Control section with macro call
PROG2       | START     |
            | EXTRN     | EXNAME
            | .         |
            | .         |
            | EXTYP2    | MNAME

* Generated instruction statements
PROG2       | START     |
            | EXTRN     | EXNAME
            | .         |
            | .         |
MNAME       | MVC       | ...
```

Type of MNAME: I
Type of &PAR: I
Type of PROG2: J
Type of EXNAME: T

Note

The following applies with respect to all previously named type attributes:
Names which are defined more than once are reported as errors and receive the type attribute of the first definition.

- The following type attributes may only have symbolic parameters which have not been replaced by the name of an instruction statement in the macro instruction.

  N In the macro call, the symbolic parameter is assigned:
  - a self-defining term,
  - a SETA symbol, or
  - a SETB symbol

  O No value is assigned to the symbolic parameter in the macro call, and the corresponding operand is omitted.

Example

```
Name          | Operation | Operands
--------------------------------------------------
* Macro definition 1
              | MACRO     |
              | EXTYPO    | &PAR1,&PAR2
&A            | SETA      | 2
              | AIF       | (&PAR1 EQ 5).SYMO
              | .         |
              | .         |
.SYMO         | ANOP      |
              | EXTYPI    | &A                  (01)
              | MEND      |

* Macro definition 2
              | MACRO     |
              | EXTYPI    | &PARIN              (02)
              | AIF       | (T'&PARIN EQ 'N').SYMI
              | .         |
              | .         |
.SYMI         | ANOP      |
              | MEND      |

* Macro call
              | EXTYPO    | 5                   (03)
```

(01) EXTYPI instruction call; the symbolic parameter &PARIN from the prototype statement of the inner macro (02) is assigned the SETA symbol &A as a value.

(03) The symbolic parameter &PAR1 is assigned the decimal self-defining term 5.

```
Type of &PAR1:   N
Type of &PARIN:  N
Type of &PAR2:   O, omitted operand
```

- The type attribute U (undefined) is given to names or variable symbols that are still undefined at interrogation time or which cannot be assigned to any of the above-mentioned types.

  The following, in particular, have type attribute U:

  – names
    – which denote a LTORG instruction, or
    – which denote an EQU instruction without a third operand,
  – SETC symbols whose value is not a name,
  – system variable symbols, except &SYSLIST(n),
  – literals as operands of macro calls.

5.5.8.2 L' Length attribute reference

The length attribute of a name or of a variable symbol is a numeric value, which represents the length of the designated memory area in bytes (for examples, see I' Integer Attribute Reference).

The length attribute reference may only be used as an element of arithmetic macro expressions.

The length attribute is 1
- in names or variable symbols which denote an EQU instruction without length specification (see section 4.2, EQU instruction) and
- in names or variable symbols whose type attribute is J, T or N.
- when L'* is used in DC and DS instructions and in literals.

The length attribute is 0
- in names or variable symbols whose type attribute is M, O or U.

The length attribute * (L'*) is the same as the length of the instruction statement in which the reference appears.

Examples

```
Name        | Operation | Operands
------------|-----------|-----------
&A          | SETB      | (L'&B EQ 4)
&C          | SETA      | L'&X+30
```

5.5.8.3 S' Scaling attribute reference

The scaling attribute of a name or variable symbol is a numeric value which specifies the number of positions in the fractional part of fixed-point, floating-point and decimal constants (for examples, see I', Integer Attribute Reference).

The scaling attribute can only be queried for names which denote the constant types mentioned, and for a symbolic parameter or SETC symbol whose value is the name of a corresponding constant.

The scale modifier reference may only appear as an element of arithmetic macro expressions.

The scaling attribute of a **fixed or floating-point number** corresponds to the value of the scale modifier.

The scaling attribute of a **decimal number** is the number of digits to the right of the decimal point.

5.5.8.4 I' Integer attribute reference

The integer attribute of a name or variable symbol is a numeric value which refers to the integer part of fixed-point, floating-point, and decimal numbers in object code. It is calculated from the length and scaling attributes.

The integer attribute reference may only be interrogated for names and symbolic parameters which denote the constant types mentioned, and for a SETC symbol whose value is the name of a corresponding constant.

The integer attribute reference may only appear as an element of arithmetic macro expressions.

Calculation of the integer attribute

Type Attrib.	Integer Attribute
H, F, G	I = 8*L-S-1
D, E, L, K	I = 2*(L-1)-S
P	I = 2*L-S-1
Z	I = L-S

For all formulas, L = length attribute
S = scaling attribute

Examples

```
Name      Operation  Operands
CONF1     DC         HS6'-15.75'          L = 2, S = 6, I = 9
CONF2     DC         FS8'100.3E-2'        L = 4, S = 8, I = 23
*
CONFL1    DC         ES2'46.415'          L = 4, S = 2, I = 4
CONFL2    DC         DS5'-3.729'          L = 8, S = 5, I = 9
*
COND1     DC         P'+1.25'             L = 2, S = 2, I = 1
COND2     DC         P'79.68'             L = 3, S = 2, I = 3
COND3     DC         Z'-543'              L = 3, S = 0, I = 3
COND4     DC         Z'79.68'             L = 4, S = 2, I = 2
```

5.5.8.5 K' Count attribute reference

The count reference of a variable symbol is a numeric value.

The count attribute reference may only be used as an element of arithmetic macro expressions.

Count attribute of symbolic parameters

The count attribute of a symbolic parameter corresponds to the number of characters of the corresponding operand in the macro call. The count attribute of an omitted operand is 0.

If the operand is a sublist, the count attribute includes the parentheses and commas of the sublist. The count attribute of each suboperand can be referenced, regardless of the nesting level (for examples, see N' Number Attribute Reference).

If the operand of a macro call contains variable symbols, the count attribute corresponds to the number of characters after the variable symbols have been replaced by their actual values.

Count attribute of SET symbols and system variable symbols

SETA symbol	Number of characters required to specify the actual value as a decimal number without leading zeros.

Examples

```
&A1 SETA 111           K of &A1 = 3
&A2 SETA X'FF'         K of &A2 = 3
&A3 SETB (K'&A2 EQ 3)  value of &A3 = 1
```

SETB symbol	1
SETC symbol	number of characters
&SYSDATE	9
&SYSECT	number of characters
&SYSLIST(n[,m])	number of characters
&SYSLIST	invalid
&SYSMOD	2
&SYSNDX	4
&SYSPARM	number of characters
&SYSTEM	4
&SYSTIME	6
&SYSTSEC	number of characters
&SYSVERM	6
&SYSVERS	6

5.5.8.6 N' Number attribute reference

The number attribute is a numeric value and may be interrogated for a symbolic parameter, for a SET symbol, or for the entire operand entry of a macro call.

The number attribute reference may only be used as an element of arithmetic macro expressions.

Number attribute of a symbolic parameter

The number attribute of a symbolic parameter corresponds to the number of suboperands in the corresponding operand sublist in the macro call. The number attribute of each suboperand in the operand sublist can be referenced.

Via the system variable symbol &SYSLIST, positional operands in a macro call, which have no corresponding symbolic parameter in the prototype statement, can also be referenced.

If the symbolic parameter accessed does not correspond to any sublist, but only to a single operand, the number attribute is 1.
If there is no operand in the macro call, the number attribute is 0.

Number attribute of SET symbols

The number attribute of a SET symbol corresponds to its actual dimension (see section 6.2, "Subscripted SET symbols").

Number attribute of &SYSLIST

The number attribute of the system variable symbol &SYSLIST corresponds to the number of positional operands in the operand entry of a macro call.

Examples

```
Name          | Operation | Operands
--------------------------------------------------
* Prototype statement
              | MAC       | &P1,&P2,&P3,&P4

* Macro call
              | MAC       | 15,'NAME',ADR,(X,(Y,Z))
```

Reference to (K'.., N'..)		Value	K	N
&SYSLIST			-	4
&SYSLIST(1)	≙ &P1	15	2	1
&SYSLIST(1,2)	≙ &P1(2)	''	0	0
&SYSLIST(2)	≙ &P2	'NAME'	6	1
&SYSLIST(3)	≙ &P3	ADR	3	1
&SYSLIST(4)	≙ &P4	(X,(Y,Z))	9	2
&SYSLIST(4,1)	≙ &P4(1)	X	1	1
&SYSLIST(4,2)	≙ &P4(2)	(Y,Z)	5	2
&SYSLIST(4,2,1)	≙ &P4(2,1)	Y	1	1

5.5.8.7 D' Definition attribute reference

The definition attribute indicates whether or not a name, which may have resulted through replacement of a symbolic parameter, has already been defined.

The definition attribute reference may be used as an element in relational expressions, in Boolean expressions, and in arithmetic macro expressions.

The definition attribute has the value 1 or 0.

1 The name or symbolic parameter referenced is defined.
0 The name or symbolic parameter referenced is not yet defined.

When using the definition attribute reference in an arithmetic macro expression, the arithmetic values +1 or +0 are used.

The definition attribute reference can be used, for example,
- in loop processing, to enquire whether a name has already been defined and execute (or not execute) the definition accordingly (see examples under Macro Definition 1);
- to enquire whether a name has already been defined and accordingly determine whether the attributes can be referenced (see examples under Macro Definition 2).

Examples

```
Name          | Operation | Operands
--------------+-----------+-----------------------------------------------------
* Macro definition 1
              | MACRO     |
              | .         |
              | .         |
              | AIF       | (D'NAME).SYM1
NAME          | DC        | F'0'
              | .         |
              | .         |
.SYM1         |           |
              | SR        | R1,R1
              | AIF       | (T'NAME EQ 'F').FCONST,(T'NAME EQ 'H').HCONST
              | IC        | R1,NAME
              | AGO       | .END
HCONST        | ANOP      |
              | LH        | R1,NAME
              | AGO       | .END
FCONST        | ANOP      |
              | L         | R1,NAME
.END          | MEND      |

* Macro definition 2
              | MACRO     |
              | .         |
              | .         |
              | AIF       | (D'NAME).SYM1
&TYP          | SETC      | 'U'
              | AGO       | .SYM2
              | .         |
              | .         |
.SYM1         | ANOP      |
&TYP          | SETC      | T'NAME
.SYM2         | ANOP      |
              | .         |
              | .         |
              | MEND      |
```

6 Variable symbols

6.1 Symbolic parameters

Symbolic parameters are defined in the prototype statement, and may then be used in name, operation and operand entries of model statements in a macro definition. When calling the macro, symbolic parameters must be assigned actual values. A symbolic parameter can be generated using one or more variable symbols, except in the prototype statement (see section 5.5.4, "Generated variable symbols").

Symbolic parameters are local symbols, i.e. are only recognized within a macro generation, since they are assigned new values in each macro call.

Keyword and positional operands

Symbolic parameters are defined in the prototype statement in the name and operand entries. The type of operand entry in the prototype statement determines whether a keyword or positional operand is involved (see example and section 7.1.1, "Keyword and positional operands").

- **Keyword operands** are indicated in the prototype statement by an equals sign (=). They can be assigned an initial value in the prototype statement. In the macro instruction, the actual values are assigned via the keyword. The sequence of keyword operands in the macro instruction is arbitrary.

  The keyword in the macro instruction is the name of the variable symbol in the prototype statement without the ampersand, or a variable symbol from which the keyword is generated.

- **Positional operands** are assigned current values in the macro instruction based on their position in the operand entry. In other words, symbolic parameters and the current values assigned must be in the same sequence both in the prototype statement and in the macro call.

Rules

- A symbolic parameter may consist of up to 64 characters.
- The first character must be an ampersand (&).
- Symbolic parameters may not begin with the character string &SYS. (see also section 6.3, "System variable symbols").
- Space characters are not allowed in a symbolic parameter.
- The same variable symbol may not be defined as a symbolic parameter and a SET symbol in the same macro definition.

Example

```
Name        | Operation | Operands
------------------------------------------------
* Macro definition
            | MACRO     |
&NAME       | MSYM      | &PAR1,&PARA=,&PARB=          (01)
&NAME       | EQU       | *
            | MVC       | &PARB,&PARA
            | B         | &PAR1
            | .         |
            | .         |
&PAR1       | EQU       | *
            | MEND      |

* Macro call
BEGIN       | MSYM      | STOP,PARA=CONST,PARB=FIELD   (02)

* Generated instruction statements
BEGIN       | EQU       | *
            | MVC       | FIELD,CONST                  (03)
            | B         | STOP                         (04)
            | .         |
            | .         |
STOP        | EQU       | *                            (05)
```

(01) Prototype statement; &PAR1 is a positional operand, &PARA and &PARB are keyword operands.

(02) Macro call; &PAR1 is assigned the current value STOP; &PARA is assigned the value CONST via the keyword PARA, and &PARB the value FIELD via the keyword PARB.

(03), (04), (05)
In these instruction statements, the symbolic parameters are replaced by the current values assigned.

6.2 SET symbols

SET symbols are variable auxiliary fields, which may be assigned values at assembly time using SET instructions (see section 7.2). They are defined either explicitly with a GBLx or LCLx instruction (see 7.2) or, for local SET symbols, implicitly via a SET instruction (see 7.2). SET symbols may be used in name, operation and operand entries of model statements. With an explicit definition, the SET symbol receives the initial value zero, or null string. These may be subsequently altered by using SET instructions, and new field contents may be assigned to the SET symbols. The SET symbols in model statements are then replaced by their current values.

SET symbols may be generated using one or more additional variable symbols (see section 5.3.3, "Generated variable symbols"). The use of SET symbols is not restricted to macros. Local or global definitions and the use of such auxiliary fields are possible in the entire source program (see chapter 8).

If SET symbols are explicitly defined, the definition must be processed before they are used for the first time. In the interest of transparency, it is advisable to process all definitions immediately after the relevant prototype statement.

Both global and local SET symbols may have an arithmetic, binary, or character value as field contents. This is differentiated beforehand in the GBLx or LCLx instruction.

The x in both instructions can be replaced by the following characters (see section 7.2, GBLx and LCLx instructions):

A SET symbol with an arithmetic value (SETA symbol)
B SET symbol with a binary value (SETB symbol)
C SET symbol with a character string as its value (SETC symbol)

Rules

– A SET symbol may consist of a maximum of 64 characters.
– The first character must be an ampersand (&).
– SET symbols may not begin with the character string &SYS (see also 6.3, System Variable Symbols).
– Space characters are not permitted in SET symbols.

Programming notes

If a SET instruction includes the name of a variable defined in the source program in the form *name DS CL(A-B)*, ASSEMBH reports a semantic error (E35) if this definition comes after the macro call. ASSEMBH will accept the SET instruction if this definition comes before the macro call or if the computed valued is used instead of the bracketed CL expression.

Predefined SET symbols

Predefined SET symbols may be assigned values without prior declaration.

SETA symbols, global: &AGm
local: &ALm with 0 ≤ m ≤ 99

SETB symbols, global: &BGn
local: &BLn with 0 ≤ n ≤ 999

SETC symbols, global only: &CGm with 0 ≤ m ≤ 99

Note

Predefined SET symbols are used by various tools and supported by ASSEMBH only on grounds of compatibility. It is therefore not advisable to use them in the creation of new programs.

Global and local SET symbols

Global SET symbols are defined with the GBLx instruction. They may be used to transfer values between macro definitions or between the macro definition and source program. In the latter instance, the symbols must also be defined in the source program.

A global SET symbol and its value is recognized in each macro definition or in the source program if it has been defined there. Redefinition does not, however, reset the initial value to zero.

Local SET symbols may be defined using the LCLx instruction. They are then only recognized in the relevant macro definition or in the source program.

If the same SET symbol is also defined as local in another macro definition, it is regarded as a separate SET symbol in each macro definition. Its respective value is not recognized in other macro definitions, including an inner one.

In local SET symbols, implied declaration is possible. No LCLx instruction is necessary in this case (see below).

Examples

```
Name          | Operation | Operands
--------------+-----------+-------------------------
* Macro definition 1
              | MACRO     |
&NAME         | LOAD1     |
              | GBLA      | &A                  (01)
*             |           |
&NAME         | LR        | 15,&A
&A            | SETA      | &A+1                (02)
              | MEND      |

* Macro definition 2
              | MACRO     |
              | LOAD2     |
              | GBLA      | &A                  (01)
*             |           |
              | LR        | 15,&A
&A            | SETA      | &A+1                (02)
              | MEND      |

* Macro calls
BEGIN         | LOAD1     |
              | LOAD2     |
              | LOAD1     |
              | LOAD2     |

* Generated instruction statements
              | .         |
              | .         |
BEGIN         | LR        | 15,0
              | LR        | 15,1
              | LR        | 15,2
              | LR        | 15,3
              | .         |
              | .         |
```

(01) Definition of the global SETA symbol &A; &A must be defined in each macro definition in which the SETA symbol is to be used.

(02) As &A was defined as global, addition is not based on the initial value 0, but on the result of the previous addition, regardless of the macro definition in which this occurred.

In the following example, &A is defined as local in both macro definitions. The other instruction statements are the same as in the first example. Here, the definition of the SETA symbol assigns the initial value 0 for each macro instruction; therefore. the SET instruction no longer affects the value of &A in the LR instruction.

```
Name        | Operation | Operands
------------------------------------
* Macro definition 1
            | MACRO     |
&NAME       | LOAD1     |
            | LCLA      | &A
*           |           |
&NAME       | LR        | 15,&A
&A          | SETA      | &A+1
            | MEND      |

* Macro definition 2
            | MACRO     |
            | LOAD2     |
            | LCLA      | &A
*           |           |
            | LR        | 15,&A
&A          | SETA      | &A+1
            | MEND      |

* Macro calls
BEGIN       | LOAD1     |
            | LOAD2     |
            | LOAD1     |
            | LOAD2     |

* Generated instruction statements
            | .         |
            | .         |
BEGIN       | LR        | 15,0
            | LR        | 15,0
            | LR        | 15,0
            | LR        | 15,0
            | .         |
            | .         |
```

Implicitly declared local SET symbols

Local SET symbols are already regarded as declared once they occur in the name entry of a SET instruction. The assembler interprets each non-declared SET symbol in the name entry of a SET instruction as a local SET symbol.

Implicitly declared local SET symbols are assigned the value specified in the operand entry of the SET instruction as the initial value. In this case, the type of SET symbol involved must be specified in the operation entry of the SET instruction (see section 7.2, SETA, SETB and SETC instructions).

Subscripted SET symbols

Global and local SET symbols may be defined as subscripted SET symbols.

Format of subscripted SET symbols &par(d)

&par is the name of the SET symbol under which d fields are accessed successively.

d dimension; $1 \leq d \leq 2^{31}-1$

- In the definition of the SET symbol (GBLx or LCLx), "d" specifies the number of fields which are to appear in succession under the name &par.
  Here, "d" may only be a decimal self-defining term.
- Using SET instructions, the individual fields may be assigned values. In the name entry of SET instructions, and when used in other model statements, "d" denotes the field which is to be referenced.
  In SET instructions, a value for "d" may be specified which is greater than the value of "d" in the definition. In that case, the value of "d" in the definition is replaced by the higher value, and this then serves as the dimension of the SET symbol.
  Here, "d" is an arithmetic macro expression.

The dimension of a SET symbol can be queried via the number attribute:

`N'&par` yields the current dimension.

Subscripted SET symbols, like other variable symbols, can be chained with variable symbols or with alphanumeric characters.

Example

```
Name          | Operation | Operands
--------------+-----------+-----------------
* Definition
              | LCLC      | &PARAM(20)

* Allocation
&PARAM(1)     | SETC      | 'FIRST'
&PARAM(2)     | SETC      | 'SECOND'
              | .         |
              | .         |
```

6.3 System variable symbols

System variable symbols are assigned values by the assembler. The programmer can use them in the name, operation, and operand entries of instruction statements but can assign no new values to them, except in &SYSMOD and &SYSLIST(n). System variable symbols can be generated by using one or more variable symbols (see section 5.3.3, "Generated variable symbols").

System variable symbols may be chained with alphanumeric characters. The syntax rules applicable to the relevant entries must be observed.

There are system variable symbols with a local or global character:

Local system variable symbols are assigned values with every macro call, so they are limited to one specific macro definition. They may only be used in macro definitions.

Local system variable symbols are:
- &SYSECT,
- &SYSLIST,
- &SYSNDX,
- &SYSTSEC, and
- &SYSVERM.

Global system variable symbols are assigned values at the start of assembly, and their value remains constant throughout the assembly.

Global system variable symbols are:
- &SYSDATE,
- &SYSPARM,
- &SYSTEM,
- &SYSTIME, and
- &SYSVERS.

The system variable symbol &SYSMOD has a special significance here. Its default is assigned at the beginning of the assembly, but it may be altered during execution of the program.

&SYSDATE

The value of &SYSDATE is the assembly date. This value is calculated at the start of assembly and remains constant.

Value of &SYSDATE mmddyyddd

mm	month
dd	day
yy	year
ddd	number of day in the year

The type attribute of &SYSDATE is U; the count attribute is 9.

Example

In the example, the assembly date is the 20th October 1989.

```
Name          | Operation | Operands
--------------------------------------------
* Macro definition
              | MACRO     |
              | MDATE     | &ENDE
              | .         |
              | .         |
              | B         | &ENDE
              | DC        | C'&SYSDATE'
              | DS        | 0H
&ENDE         | EQU       | *
              | MEND      |

* Macro call
              | MDATE     | ENDE

* Generated instruction statements
              | .         |
              | .         |
              | B         | ENDE
              | DC        | C'102089293'
              | DS        | 0H
ENDE          | EQU       | *
              | .         |
              | .         |
```

&SYSECT

The value of &SYSECT is the name of the current control section at the time of the macro instruction. Additional information on &SYSECT is provided by the system variable symbol &SYSTSEC, which contains the corresponding type of each control section (see description of &SYSTSEC).

If &SYSECT occurs in a macro definition, its value is the name of the last START, CSECT, DSECT, XDSEC or COM instruction processed prior to the macro call. Whether or not the instruction was correct is ignored, provided the error has not resulted in the control section not being defined.

For the relevant macro level, the value of &SYSECT is constant during processing of a macro definition, regardless of CSECT, DSECT, XDSEC or COM instructions or inner macro calls.

If, on the other hand, CSECT, DSECT, XDSEC or COM instructions appear in a macro definition, they affect the value of &SYSECT for all subsequent inner macro instructions in this macro definition and for all subsequent macro calls at another macro level.

The type attribute of &SYSECT is U; the count attribute corresponds to the number of characters which represent the value of &SYSECT.

Example

```
Name        | Operation | Operands
------------------------------------------
* Macro definition, outer macro
            | MACRO     |
            | MACA      | &NAME,&CSECT
&NAME       | DC        | C'&SYSECT'
&CSECT      | CSECT     |
            | DC        | C'&SYSECT'
            | MACI      |
            | MEND      |

* Macro definition, inner macro
            | MACRO     |
            | MACI      |
            | DC        | C'&SYSECT'
            | MEND      |

* Control section with macro calls
PROG        | START     |
            | .         |
            | .         |
            | MACA      | FIRST,ACSECT
**************************************
            | MACA      | SECOND,BCSECT

* Generated instruction statements
FIRST       | DC        | C'PROG'           (01)
ACSECT      | CSECT     |
            | DC        | C'PROG'           (01)
            | DC        | C'ACSECT'         (02)
**************************************
SECOND      | DC        | C'ACSECT'         (03)
BCSECT      | CSECT     |
            | DC        | C'ACSECT'         (03)
            | DC        | C'BCSECT'         (04)
```

(01) The two instructions are at the same macro level, and the macro was called in PROG.

(02) The DC instruction originates from the instruction of the inner macro MACI. The MACI call is only processed after the CSECT instruction ACSECT, and the value of &SYSECT is therefore ACSECT.

(03) The two instructions are again at the same macro level, but the last preceding CSECT instruction was ACSECT. The value of &SYSECT in both instructions is therefore ACSECT.

(04) The instruction again comes from the inner macro instruction MACI. This is another macro level, and &SYSECT thus assumes the name of the preceding CSECT instruction BCSECT.

&SYSLIST

&SYSLIST can be used to reference positional operands of macro calls instead of symbolic parameters. Using the &SYSLIST index, every positional operand, and in particular, every sublist in a macro call can be referred to, even if no corresponding symbolic parameter has been defined in the prototype statement.

The attributes of operands in a macro call can also be referenced in this way.

```
Format of &SYSLIST    &SYSLIST(n[,m[,...]])
```

n denotes the nth positional operand in a macro call.

If n = 0, the result is the name entry of the macro call.

m denotes the mth operand of the operand sublist which is the nth operand in a macro call.

Subscripting can be further extended to refer to each suboperand in an operand sublist (see section 7.1.2, "Operand sublists").

n or m may be any positive arithmetic expression which may be an operand in a SETA instruction.

If there is no operand for a value of n or m, the result is a blank character string.

Only with a number attribute reference of the operand entry in a macro call (N'&SYSLIST, see section 5.5.8, "Attribute references") can &SYSLIST be used without the index n or m. In this instance, the result is the number of positional operands in the macro call.

The type attribute reference (T'&SYSLIST(n)) gives the type of operand accessed.

Example

```
Name        | Operation | Operands
------------------------------------------------
* Macro definition
            | MACRO     |
            | MLIST     | &PAR
&PAR        | DC        | C'&SYSLIST(2)'
            | DC        | C'&SYSLIST(3,2)'
            | DC        | C'&SYSLIST(4)'
            | DC        | C'&SYSLIST(2,1)'
            | DC        | C'&SYSLIST(3)'
            | MEND      |

* Macro call
            | MLIST     | AAA,BBB,(C,D,,F),,H

* Generated instruction statements
AAA         | DC        | C'BBB'
            | DC        | C'D'
            | DC        | C''                    blank char.string
            | DC        | C'BBB'
            | DC        | C'(C,D,,F)'
```

&SYSMOD

&SYSMOD can be used to control the resolution of a mode-related system macro, depending on the addressing mode (see section 3.3.3).

At the beginning of the assembly, &SYSMOD is allocated the default 24. The value of &SYSMOD can be altered via the system macro GPARMOD (see manual "Introductory Guide to XS Programming" [7]).

For SYSMOD, only the values 24 and 31 are valid.

The type attribute of &SYSMOD is U, the count attribute is 2.

In the resolution of mode-related system macros, the value of &SYSMOD is used as a default for the global parameter mode (see PARMOD and GPARMOD in "Introductory Guide to XS Programming" [7]).

&SYSNDX

The value of &SYSNDX is a counter, which is incremented by 1 with each inner or outer macro instruction processed in an assembly unit. The counter has four positions, and is set to 0001 for the first macro instruction, to 0002 for the second, etc.

Since the value of &SYSNDX does not represent any valid name, &SYSNDX must chained with a valid name if &SYSNDX is to be used for the generation of names.

The value of &SYSNDX is constant during processing of a macro definition, regardless of any inner macro calls. &SYSNDX can therefore be used to generate unique names for instruction statement strings which have been generated by multiple calls in the same macro definition.

If &SYNDX appears as an element in an arithmetic expression, its value is interpreted arithmetically.

The type attribute of &SYSNDX is U, and the count attribute is 4.

Example

```
Name          | Operation | Operands
--------------------------------------------------
* Macro definition, inner macro
              | MACRO     |
              | MACI      | &PARAM
A&SYSNDX      | SR        | 2,5
              | CR        | 2,5
              | BE        | B&PARAM
              | B         | A&SYSNDX
              | MEND      |

* Macro definition, outer macro
              | MACRO     |
&NAME         | MACA      | &PARAM
&NAME         | SR        | 2,4
B&SYSNDX      | AR        | 2,6
              | MACI      | &PARAM
B&PARAM       | S         | 2,=F'1000'
A&SYSNDX      | ST        | 2,WORD
              | MEND      |

* 1st MACA call
ALPHA         | MACA      | XXX                (01)

* Generated instruction statements
ALPHA         | SR        | 2,4
B0106         | AR        | 2,6                (02)
              | MACI      | XXX                (04)
A0107         | SR        | 2,5                (05)
              | CR        | 2,5
              | BE        | BXXX
              | B         | A0107              (06)
BXXX          | S         | 2,=F'1000'
A0106         | ST        | 2,WORD             (03)

* 2nd MACA call
BETA          | MACA      | YYY                (07)

* Generated instructions
BETA          | SR        | 2,4
B0108         | AR        | 2,6                (08)
              | MACI      | YYY                (10)
A0109         | SR        | 2,5                (11)
              | CR        | 2,5
              | BE        | BYYY
              | B         | A0109              (12)
BYYY          | S         | 2,=F'1000'
A0108         | ST        | 2,WORD             (09)
```

(01) This instruction statement is to be the 106th macro instruction in this assembly unit. A&SYSNDX is replaced by A0106, and B&SYSNDX by B0106 in instructions (02) and (03).
(04) 107th macro instruction, A&SYSNDX is replaced by A0107; see instruction statements (05) and (06).
(07) 108th macro call, A&SYSNDX and B&SYSNDX are replaced by A0108 and B0108; see instruction statements (08) and (09).
(10) 109th macro call, A&SYSNDX is replaced by A0109; see instruction statements (11) and (12).

&SYSPARM

The value of &SYSPARM is a string of up to 255 characters. These are defined in an SDF option (see "ASSEMBH User Guide" [1]) and evaluated in the macro resolution; i.e. an option entered externally can be used to control the processing of certain instruction statement strings.

The type attribute of &SYSPARM is U; the count attribute corresponds to the number of characters defined.

&SYSTEM

The value of &SYSTEM is the operating system version under which the assembly runs.

```
Value of &SYSTEM     2vvv
```

2 stands for BS2000
vvv version designation of BS2000

The type attribute of &SYSTEM is U; the count attribute is 4.

&SYSTIME

The value of &SYSTIME is the time of day of the assembly. This value is calculated at the beginning of the assembly and remains constant.

```
Value of &SYSTIME     hhmmss
```

hh hours
mm minutes
ss seconds

The type attribute of &SYSTIME is U; the count attribute is 6.

&SYSTSEC

The value of &SYSTSEC is the type of the current control section at the time of the macro instruction. In other words, &SYSTSEC contains the the type of control section whose name is stored in &SYSECT (see description of &SYSECT).

Control section	Value of &SYSTSEC
START	CSECT
CSECT	CSECT
DSECT	DSECT
COM	COM
XDSEC	XDSEC

If &SYSTSEC appears in a macro definition, its value is the type of the last START, CSECT, DSECT, COM or XDSEC instruction processed prior to the macro call. Whether or not the instruction was correct is ignored, provided the error has not resulted in the control section not being defined.

The type attribute of &SYSTSEC is U; the count attribute corresponds to the number of characters which make up the value of &SYSTSEC.

Example

```
Name        | Operation | Operands
------------------------------------------------------
* Macro definition
            | MACRO     |
            | MACDSECT  | &NAME
&NAME       | DSECT     |
            | DS        | 4F
&SYSECT     | &SYSTSEC  |                  Reset to the original control
            | MEND      |                  section

* Control section with macro instruction
PROG        | START     |
            | .         |
            | .         |
            | MACDSECT  | ADSECT
            | .         |
            | .         |

* Generated instruction statements
ADSECT      | DSECT     |
            | DS        | 4F
PROG        | CSECT     |                  Continuation of 1st control
            | .         |                  section
            | .         |
```

&SYSVERM

The value of &SYSVERM is the version designation of the library macro in which this parameter is used.

Macro definitions in the source program have no version designation; thus, it makes no sense to use &SYSVERM in this case. Here, the version designation is replaced by space characters (VER␣␣␣).

```
Value of &SYSVERM     VERvvv
```

vvv version designation

A version designation which is too long is truncated to the right, while one which is too short is padded to the right with underscores (_).

The type attribute of &SYSVERM is U; the count attribute is 6.

Examples

Version designation	002	gives	VER002
Version designation	2	gives	VER2__
Version designation	1234	gives	VER123

&SYSVERS

The value of &SYSVERS is the version designation of the source program, provided this is available in a library.

Source programs which are not read from a library have no version designation; it therefore makes no sense to use &SYSVERS here. The version designation is in this case replaced by space characters (VER␣␣␣).

```
Value of &SYSVERS     VERvvv
```

vvv version designation

A version designation which is too long is truncated to the right, one too short is padded to the right with underscores (_) (see &SYSVERM for examples).

The type attribute of SYSVERS is U; the count attribute is 6.

7 Macro language instructions

7.1 Prototype statement and macro call

Function

The prototype statement specifies the name of the macro and the symbolic parameters appearing in this macro definition.

The macro call is an instruction in the assembler source program. Processing of the macro call initiates macro generation. Through this, the instruction statements preset by the macro definition are inserted into the assembler program by means of variable symbols. The macro call assigns the relevant current values to the symbolic parameters of the prototype statement.

Format of the prototype statement

Name	Operation	Operands
[&par]	name	[{pos_oper[,...] / keyw_oper[,...] / pos_oper[,...],keyw_oper[,...]}]

&par Symbolic parameter
name Macro name

pos_oper Positional operand; format: &par
&par symbolic parameter

keyw_oper Keyword operand; format: &par=[vala]
&par symbolic parameter
vala initial value

Several positional and keyword operands may be mixed in a prototype statement. They may appear in any order.

Format of the macro instruction

Name	Operation	Operands
[{valn / .sym / &par1}]	{name / &par2}	[{pos_oper[,...] / keyw_oper[,...] / pos_oper[,...],keyw_oper[,...]}]

valn — Name which is to be allocated to the symbolic parameter in the name entry of the prototype statement

.sym — Sequence symbol

&par1 — Variable symbol or concatenation of variable symbols and alphanumeric characters; the name entry must be generated from this.
The value of &par1 is allocated to the symbolic parameter &par in the name entry of the prototype statement.

name — Macro name

&par2 — Variable symbol or concatenation of variable symbols and alphanumeric characters; the value corresponds to the macro name.

pos_oper — Positional operand; format: val

- val — Value which is to be allocated to the symbolic parameter of the prototype statement
  or variable symbol

keyw_oper — Keyword operand; format: par=val

- par — Keyword, name of the symbolic parameter of the prototype statement (without ampersand)
  or variable symbol, where the value must be a keyword.
- var — Value which is to be allocated to the symbolic parameter
  or variable symbol

Several positional and keyword operands may be mixed in the macro call. Keyword operands may be in any order, but positional operands must be in the same sequence as the positional operands in the prototype statement (for example, see section 7.1.1).

Name entry

The name entry of the macro call is assigned to the symbolic parameter in the name entry of the prototype statement.

If the name entry of the macro call contains a variable symbol, this must
- be defined in the source program if the macro call is in the source program, or
- be defined in the macro definition which contains the macro call.

Operation entry

The operation entry of the prototype statement specifies the name with which the macro must be called. This must be created in accordance with the rules for names (see section 2.3, "Name entry"). In the case of a macro definition in a library, it must also adhere to LMS syntax (see the "LMS" User Guide [4]).

If there are two or more macro definitions with the same name in the operation entry in a source program, the last macro definition read in is regarded as valid until the next macro definition of the same name appears.

For a macro definition in the source program, the mnemonic operation code of an instruction statement can be used in the operation entry of the prototype statement, without canceling the instruction statement with the OPSYN instruction. The macro definition must be executed prior to its first instruction. Here, the original function of the instruction statement is deactivated, and every time the instruction statement appears, it is regarded as the invocation of the corresponding macro.
Note that no further macros can be read after processing a macro definition in the source program with the operation code ’MACRO’ in the prototype statement.

If a macro definition in a library receives the mnemonic operation code of an assembler instruction statement as a name, the corresponding operation code must first be canceled by using the OPSYN instruction.

If the operation entry of the macro call contains a variable symbol it must
- be defined in the source program if the macro call is in the source program or
- be defined in the macro definition in which the macro call appears.

Operand entry

The operand entry of the prototype statement determines the format and structure of the operand entry of the relevant macro call. The macro call assigns current values to the symbolic parameters of the prototype statement.

If an operand is omitted in the macro call, the corresponding symbolic parameter is assigned a null string (no characters) as the character value, or the arithmetic value zero. Keyword parameters that have been assigned an initial value in the prototype statement are exceptions.

Rules for macro instruction operands

- A macro instruction operand may be up to 1020 characters in length, even after any variable symbols have been replaced. In keyword operands too, the value must be 1020 characters long. The keyword and the equals sign are not counted here.
- Single quotes in an operand must appear in pairs.

  A single single quote in a character string must be represented by two single quotes.

  A length attribute reference (L') of a name is permitted as an operand.
- Left and right parentheses must appear in pairs, unless enclosed in single quotes. If an operand begins with a left parenthesis, it is interpreted as a sublist. If the closing right parenthesis is not followed by a space or comma, the entire operand is interpreted as a character string and not as a sublist.
- If several ampersands appear one after the other, their number must be even, except if an ampersand indicates a variable symbol.
- A comma denotes the end of an operand or sublist element, unless it is enclosed in single quotes.
- A space character indicates the end of the operand entry, unless it is enclosed in single quotes.
- If the operand of the macro call is a macro expression, the corresponding symbolic parameter will be assigned the attributes of a character constant.
- If a macro instruction operand contains a variable symbol as its value, this symbol must
  - be defined in the source program if the macro call is in the source program, or
  - be defined in the macro definition in which the macro call appears.

7.1.1 Keyword and positional operands

Keyword operands are identified in the prototype statement by the equals sign (=). They can be assigned an initial value in the prototype statement.

In the macro call, keyword operands are assigned values via the keyword. Keyword operands may appear in any order in the macro call.

No comma may be specified if a keyword operand is omitted in the macro call.

For a keyword operand in the macro call, an additional variable symbol can be used to generate the value to be assigned.

Positional operands are assigned on the basis of their position in the operand entry. Symbolic parameters and the current values assigned must be in the same order in both the prototype statement and the macro instruction.

If no value is assigned to a positional operand in a macro call, it is omitted, and the subsequent separating comma is entered. If the last positional operand in the macro call is omitted, the corresponding commas may also be left out.

The value to be assigned to a positional operand in the macro call may be generated using a variable symbol.

More positional operands may appear in the macro call than specified in the prototype statement. If this is the case, the extra operands may only be referenced via the system variable symbol &SYSLIST (see section 6.3).

Keyword and positional operands may appear mixed in the prototype statement and macro call. In positional operands, however, the order must be the same in both the prototype statement and the macro call.

Examples

```
Name        | Operation | Operands
------------------------------------------------
* Prototype statement
&NAME       | MAC       | &P1,&P2=VAL2,&P3,&P4=,&P5

* Macro call 1
VAL         | MAC       | P4=VAL4,VAL1,VAL3,VAL5
```

&P1,&P3 and &P5	Positional operands; these are assigned the values VAL1 VAL3 and VAL5.
&P2	Keyword operand with the initial value VAL2.
&P4	Keyword operand; the value VAL4 is assigned via the macro call.

```
* Macro call 2
VAL         | MAC       | P4=VAL4,VAL1,,VAL5
```

Here, no value is to be assigned to the positional operand &P3. It is replaced in the macro call by the comma and receives the value "null string".

7.1.2 Operand sublists

The value of a positional operand in a macro call or the value of a keyword operand in a macro call or prototype statement may be an operand sublist. In this case, the entire sublist is regarded as an operand of the macro call and is assigned to a symbolic parameter of the prototype statement.

```
Format of the operand sublist     (val,val[,...])
```

val	Value or new operand sublist

```
Reference to suboperands in the operand sublist     &P(n[,m[,...]])
```

&P	Symbolic parameter from the prototype statement to which the operand sublist has been assigned.
n	denotes the nth suboperand in the operand sublist and is a positive arithmetic macro expression.
m	denotes the mth suboperand in the sublist, which in turn represents the nth suboperand of a sublist and is a positive arithmetic macro expression.

Subscripting can be further extended by referring to each suboperand within any required level of the sublist.

Rules

- Sublists must always begin with "(" and end with ")"; otherwise, the operand (or sublist element) will be interpreted as a string.
- Operand sublists may be nested to any desired level; the only restriction comprises the maximum length of operands.
- The maximum length of 1020 characters for macro instruction operands refers here to the entire sublist, inclusive of suboperands, commas, and parentheses.
- As with positional operands, omitted suboperands may be replaced by a comma, and receive the value zero.
- If a macro instruction operand is referenced as a sublist in an instruction statement although it is not one, &P(1) refers to all operands. &P(n) with n > 1 then has the value zero, or null string.
- Suboperands of operand sublists can also be referenced in positional operands using the system variable symbol &SYSLIST (see section 6.3).

- If an operand sublist contains a variable symbol in the suboperand, it must be defined
  - in the source program, if the macro call is in the source program, or
  - in the macro definition in which the macro call occurs.
- Spaces are legal in operand sublists; they are interpreted as a part of a suboperand. For example, the sublist (1␣,2,␣3) has three suboperands:

  sub1 = 1␣
  sub2 = 2
  sub3 = ␣3

Example

```
Name        | Operation | Operands
------------------------------------------------------------
* Macro definition
            | MACRO     |
            | SUBEX     | &P
            | AIF       | (&P(3,2,1) EQ 'F' OR &P(5) EQ '').SYM
            | .         |
            | .         |
            | MEND      |

* Macro call
            | SUBEX     | (A,(B,C),(D,(E,F)),G,)
```

Operand in instr. (A,(B,C),(D,(E,F)),G,)

Suboperands			
	&P(1) = A		
	&P(2) = (B,C)	&P(2,1) = B &P(2,2) = C	
	&P(3) = (D,(E,F))	&P(3,1) = D &P(3,2) = (E,F)	&P(3,2,1) = E &P(3,2,2) = F
	&P(4) = G		
	&P(5) = ''		

In the AIF instruction, &P(3,2,1) is replaced by E
and &P(5) is replaced by '' (empty string).

7.1.3 Outer and inner macro instructions

A macro instruction which is used in a macro definition as a model statement is an inner macro instruction or call. The instruction in the relevant macro definition, which contains the inner instruction, is known as the outer macro instruction. The outer macro instruction may be in the source program, or may be an inner macro instruction in another macro definition.

A macro instruction in the source program is also known as the first level; the inner macro instruction in the relevant macro definition is the second level. The macro definition pertaining to the second level instruction may contain a further inner macro instruction, which is then a third level, etc.

In this way, a maximum of 255 levels can be nested. The possible nesting level which can be achieved is also dependent on the complexity of the macro definitions and on the available memory space.

Transfer of values via inner macro instructions

The values in the operand entry of the inner macro instruction replace the symbolic parameters of the corresponding prototype statement.

Values can be transferred from an outer macro to the relevant macro definition via an inner macro instruction.

The following are the conditions for this:
- The symbolic parameter of the outer macro or
- the SET symbol defined in the outer macro must be used as operands of the inner macro instruction.

The values specified as operands in the outer macro instruction then replace the corresponding symbolic parameters.

Example

```
Name          | Operation | Operands
--------------+-----------+-------------------------------------
* Macro definition 1
              | MACRO     |
&NAME         | MADD      | &ART,&F1,&F2,&F3,&F4            (02)
              | .         |
              | .         |
&NAME         | MPACK     | &F1,&F2,&F3,&F4                 (03)
              | .         |
              | .         |
&NAME         | AP        | &F1,&F2
              | AP        | &F1,&F3
              | AP        | &F1,&F4
              | MEND      |

* Macro definition 2
              | MACRO     |
&NAME         | MPACK     | &A1,&A2,&A3,&A4                 (04)
&NAME         | PACK      | &A1,&A1
              | PACK      | &A2,&A2
              | PACK      | &A3,&A3
              | PACK      | &A4,&A4
              | MEND      |

* Outer macro instruction
UNP           | MADD      | U,FIELD1,FIELD2,FIELD3,FIELD4   (01)

* Generated inner macro instruction
UNP           | MPACK     | FIELD1,FIELD2,FIELD3,FIELD4

* Generated instruction statements
              | .         |
              | .         |
UNP           | AP        | FIELD1,FIELD2
              | AP        | FIELD1,FIELD3
              | AP        | FIELD1,FIELD4
              | .         |
              | .         |
FIELD1        | DC        | CL6'000010'
              | .         |
              | .         |
```

(01) Outer macro instruction;

(02) Prototype statement of macro definition 1; the symbolic parameters are replaced by the values from the macro instruction.

(03) Inner macro instruction; the values of the outer macro instruction are transferred to the symbolic parameters in this macro instruction.

(04) Prototype statement of macro definition 2; the symbolic parameters are replaced by those of the macro instruction (03) and thus by the values transferred.

7.1.4 Alternative statement format

The alternative statement format is an additional format option for prototype statements and macro calls. It can not only be used for LCLx and GBLx instructions, but also for format 2 of AIF and AGO instructions.

Format

Name	Operation	Operand	Continuation Character
[&par]	name	operand, operand[, ...]	x x

Name entry	See description of normal format
Operation entry	See description of normal format
operand	Positional or keyword operand(s), or suboperands in the case of an operand list
x	Any character other than a space character

Description

The alternative statement format allows one or more operands to be entered in a line and continued in the next line with the next operand. This is also true for operands in operand sublists.

The following applies with respect to the operand entry:

- To enable an operand entry to be continued in the next line starting with the continue column, the preceding operand must be followed by a comma. The continuation character column must then contain a character other than a space.
- After the comma and a space character, remarks may be entered up to and including the end column.
- A space character after an operand (but not a suboperand) indicates the end of the operand entry.
- Remarks may follow even after the end of the operand entry. If a character other than a space character is in the continuation character column, the remarks entry is continued in the next line, starting with the continue column.

Example

```
Name        Operation   Operand                        Continuation Char.
                        OP1         OPER1,OPER2,OPER3      COMMENT          (01)
*
                        OP2A        OPER1,                 COMMENT  X
                                    OPER2,                 COMMENT  X
                                    OPER3                  COMMENT
*                                                                           (02)
                        OP2B        OPER1,                 COMMENT  X
                                    OPER2,OPER3,OPER4,     COMMENT  X
                                    OPER5                  COMMENT
*
                        OP3         OPER1,                 COMMENT  X
                                    OPER2                  COMMENT  X       (03)
                                    COMMENT
*
                        OP4         OPER1,OPER2            COMMENT  X
                                    (SUBOPER31,            COMMENT  X       (04)
                                    SUBOPER32)
```

(01) Normal format
(02) Alternative statement format with remarks entry
(03) Alternative statement format with remarks entry, continued in the next line.
(04) Alternative statement format with operand sublists

7.2 Description of macro statements

ACTR Count branches

Function

The ACTR instruction is used to limit AGO and AIF branches processed in a macro definition or in the assembler source program. This prevents continuous loops during macro generation.

Format

Name	Operation	Operands
[.sym]	ACTR	arexp

.sym Sequence symbol
arexp Positive arithmetic macro expression

Description

The ACTR instruction sets a counter to the value specified by arexp. The counter is decremented by one for every AGO or AIF branch.

The ACTR instruction can be set for every macro definition and for the assembler source program. The various counters are independent of each other.

If the counter is zero prior to the branch, the following results:

– For a counter which is meant for a macro definition, processing of this macro definition and any inner macro definitions contained therein is aborted, and the next instruction statement after the macro instruction is processed.

– For a counter meant for the assembler source program, assembly is aborted, and only errors previously found are reported.

Programming notes

If no ACTR instruction is specified, the assembler assumes the value of the counter to be 4096.

See example under AGO instruction.

AIF Conditional branch

Function

The AIF instruction enables conditional branching to be executed.

Format 1

Name	Operation	Operands
[.sym1]	AIF	(logexp).sym2

.sym1 Sequence symbol in standard format
.sym2 Sequence symbol in standard format or generated
logexp Boolean expression

Note

The operation code AIFB, which can be used in the same way as AIF, is supported only for reasons of compatibility.

Description

logexp is calculated as per the rules for Boolean expressions; the result may be true or false:

– If logexp is true, a branch is made to the instruction statement denoted by .sym2.
– If logexp is false, the instruction statement after the AIF instruction is processed.

Programming notes

1. The AIF instruction can be used to branch forwards or backwards. Instruction statements skipped as a result of a branch are not generated.
2. With the AIF instruction, no branch from one macro definition to another, or from source program to macro definition, and vice-versa, is possible.

Example

The example shows a macro in which the symbolic parameter from the macro instruction is checked for certain conditions. Only if all the necessary conditions have been met will processing be executed.

```
Name      | Operation | Operands
----------+-----------+-------------------------------------
          | MACRO     |
&NAME     | TAIF      | &REG,&ADR,&NR                        (01)
          | LCLB      | &ERRIN                               (02)
          | .         |
          | .         |                                    ]
          | AIF       | (&REG GE 2 AND &REG LE 13).OK1     |
          | MNOTE     | 0,'INCORRECT REG &REG'             |
&ERRIN    | SETB      | 1                                  |
.OK1      | AIF       | (&ADR NE '').OK2                   |
          | MNOTE     | 0,'MISSING ADR'                    } (03)
&ERRIN    | SETB      | 1                                  |
.OK2      | AIF       | (&NR GE 1 AND &NR LE 4).OK3        |
          | MNOTE     | 0,'INCORRECT NR &NR'               |
&ERRIN    | SETB      | 1                                  |
.OK3      | ANOP      |                                    ]
          | AIF       | (&ERRIN).MEND                        (04)
          | .         |
          | .         |
.MEND     | MEND      |
```

(01) Prototype statement with symbolic parameters &REG, &ADR and &NR

(02) Definition of an "error indicator" &ERRIN

(03) The 3 AIF inquiries check whether the symbolic parameters have been specified as per the required conditions in the macro instruction. If the parameter is correct, a branch is made to the next inquiry.
If an error occurs, the MNOTE instruction is executed; this displays a suitable explanation, and sets the error indicator.

(04) Inquiry as to whether the error indicator has been set anywhere. If so, there is a branch to the end; if not, the assembler processes the next instruction statements.

Format 2

Name	Operation	Operands
[.sym]	AIF	{(logexp).sym1}[,...]

.sym Sequence symbol in standard format
logexp Boolean expression
.sym1 Sequence symbol in standard format or generated

Description

Format 2 of the AIF instruction corresponds to n successive AIF instructions. Boolean expressions are calculated one after the other, and a branch is made to the sequence symbol whose relevant Boolean expression is true.

Programming notes

The alternative statement format (see section 7.1.4) can be used for format 2 of the AIF instruction.

Example

Name	Operation	Operand	Continuation Char.
	AIF	('&S' EQ '+').PLUS, ('&S' EQ '-').MINUS, ('&S' EQ '=').EQ	X X
	. .		

The example is shown in the alternative statement format. Depending on the contents of &S, a branch is made to the relevant sequence symbol.

AGO Unconditional branch

Function

An unconditional branch can be made using the AGO instruction.

Format 1

Name	Operation	Operands
[.sym1]	AGO	.sym2

.sym1 Sequence symbol in standard format
.sym2 Sequence symbol in standard format or generated.

Note

The operation code AGOB, which can be used in the same way as AGO, is still supported for reasons of compatibility.

Description

".sym2" denotes the next instruction statement which is to be processed by the assembler.

Programming notes

1. The AGO instruction can be used to branch forwards or backwards. Instruction statements skipped as a result of a branch are not generated.
2. With the AGO instruction, no branch from one macro definition to another, or from source program to macro definition, and vice-versa, is possible.

Example

The example shows the use of AGO in a loop, in relation to an AIF inquiry.

```
Name        | Operation | Operands
------------------------------------------------
* Macro definition
            | MACRO     |
            | REG       | &PRE
            | LCLA      | &R
            | .         |
            | .         |
            | ACTR      | 100                  (01)
.LOOP       | ANOP      |                      (02)
&PRE&R      | EQU       | &R
&R          | SETA      | &R+1
            | AIF       | (&R GT 15).END       (03)
            | AGO       | .LOOP                (04)
            | .         |
            | .         |
.END        | MEND      |

* Macro call
            | REG       | REG

* Generated instruction statements
REG0        | EQU       | 0
REG1        | EQU       | 1
REG2        | EQU       | 2
            | .         |
            | .         |
```

(01) "Emergency brake", lest loop criterion is not achieved
(02) Start of loop
(03) Inquiry as to end criterion, and branch out of loop
(04) End of loop, and return to start

Format 2

Name	Operation	Operands
[.sym]	AGO	(arexp){.sym1}[,...]

.sym Sequence symbol in standard format
arexp Positive arithmetic macro expression
.sym1 Sequence symbol in standard format or generated

Description

"arexp" specifies the number of sequence symbols in the operand entry to which the branch is to be made.

If the result of arexp is greater than the number of sequence symbols, or smaller than 1, no branch is executed.

Programming notes

The alternative statement format (see section 7.1.4) can be used in format 2 of the AGO instruction.

Example

Name	Operation	Operands
	AGO	(&EX).LOOP1,.LOOP2,.LOOP3
	.	
	.	

Here, a branch is made to .LOOP2 if &EX assumes the value 2.
If &EX has a value greater than 3, the instruction statement after the AGO instruction is executed.

In the alternative statement format, the same example is coded as follows:

Name	Operation	Operand	Continuation Char.
	AGO	(&EX).LOOP1,	X
		.LOOP2,	X
		.LOOP3	
	.		
	.		

ANOP No operation

Function

With the ANOP instruction, sequence symbols can be defined as the branch destination for a branch. It does not execute any functions.

Format

Name	Operation	Operands
[.sym]	ANOP	

.sym Sequence symbol

Description

The ANOP instruction itself does not perform any operation itself; it is merely used as a destination for branches in the macro language. The assembler generates the next respective instruction statement.

Programming notes

1. The ANOP instruction is mainly used if a branch is to be made to an instruction statement which permits no sequence symbol in the name field. In this case, the ANOP instruction must be inserted prior to the instruction statement.
2. By using ANOP for branch destinations, another instruction may be easily inserted as the first or deleted.

Example

```
Name        Operation   Operands
            AGO         .SYM
            .
            .
.SYM        ANOP                    insert
                                    <-------        AIF   (D'RDEF).SYM1
RDEF        DC          F'123'
                                    <------- .SYM1  ANOP
            AIF         (&A EQ 0).SYM2
            .
            .
```

GBLx Define global SET symbol

Function

The GBLx instruction defines one or more global SET symbols.

Format

Name	Operation	Operands
	GBL{A|B|C}	{&par|&par(d)}[,...]

&par	Global SET symbol
&par(d)	Subscripted global SET symbol
d	Dimension; decimal self-defining term
GBLA	Definition of a SETA symbol
GBLB	Definition of a SETB symbol
GBLC	Definition of a SETC symbol

Description

The GBLx instruction defines the global SET symbol or symbols in the operand entry, and assigns it/them an initial value, if required. The defined SET symbols are all of the type specified in the operation entry.

Initial values:	SETA symbol	0
	SETB symbol	0
	SETC symbol	null string

Programming notes

1. A global SET symbol is assigned an initial value only by the first GBLx instruction processed. Subsequent GBLx instructions in another macro definition or in the source program signify only the definition of the SET symbol, and do not assign the initial value again.
2. SET symbols may not begin with the character string &SYS (see section 6.3, "System variable symbols").
3. The alternative statement format (see section 7.1.4) can be used for the GBLx instruction.

Example

```
Name        | Operation | Operands
------------|-----------|-------------------------
            | GBLA      | &(&AR1),&AR2(20),&AR3,&AR4
```

&(&AR1) Global SETA symbol, whose name is to be generated using &AR1
&AR2(20) Subscripted global SETA symbol with dimension 20
&AR3, &AR4 Global SETA symbol

In the alternative statement format, the same example is coded as follows:

```
Name | Operation | Operand      | Continuation Char.
-----|-----------|--------------|-------------------
     | GBLA      | &(&AR1),     | X
     |           | &AR2(20),    | X
     |           | &AR3,        | X
     |           | &AR4         |
```

LCLx Define local SET symbol

Function

The LCLx instruction defines one or more local SET symbols.

Format

Name	Operation	Operands
	LCL[A|B|C]	{&par | &par(d)}[,...]

&par	Local SET symbol
&par(d)	Subscripted local SET symbol;
d	Dimension; decimal self-defining term
LCLA	Definition of a SETA symbol
LCLB	Definition of a SETB symbol
LCLC	Definition of a SETC symbol

Description

The LCLx instruction defines the local SET symbol or symbols in the operand entry and assigns it/them an initial value.

The defined SET symbols are all of the type specified in the operation entry.

Initial values:	SETA symbol	0
	SETB symbol	0
	SETC symbol	blank char.string

Programming notes

1. If the same SET symbol is defined as local in both the source program and in one or more macro definitions, it is regarded as a new SET symbol with the initial value in each case.
2. SET symbols may not begin with the character string &SYS (see section 6.3, "System variable symbols").
3. The alternative statement format (see section 7.1.4) can be used for the LCLx instruction.

Example

```
Name        | Operation | Operands
------------|-----------|-----------------------------
            | LCLA      | &(&AR1),&AR2(20),&AR3,&AR4
```

&(&AR1) Local SETA symbol, whose name is to be generated using &AR1
&AR2(20) Subscripted local SETA symbol with dimension 20
&AR3, &AR4 Local SETA symbol

In the alternative statement format, the same example is coded as follows:

```
Name | Operation | Operand          | Continuation Char.
-----|-----------|------------------|-------------------
     | LCLA      | &(&AR1),         | X
     |           | &AR2(20),        | X
     |           | &AR3,            | X
     |           | &AR4             |
```

MACRO Macro definition header

Format

Name	Operation	Operands
	MACRO	

Description

The MACRO (macro definition header) instruction identifies the start of a macro definition. It must be the first instruction statement in each macro definition.

MEND Macro definition trailer

Format

Name	Operation	Operands
[.sym]	MEND	

.sym Sequence symbol

Description

The MEND (macro definition trailer) instruction identifies the end of a macro definition. It must be the last instruction statement in each macro definition.

The sequence symbol .sym is used as a branch destination for AIF or AGO instructions.

MEXIT Define exit from a macro definition

Function

The MEXIT instruction defines an exit from a macro definition. During generation, this terminates the current macro generation.

Format

Name	Operation	Operands
[.sym]	MEXIT	

.sym Sequence symbol

Description

As a result of the MEXIT instruction, the assembler terminates the current macro generation, and processes the instruction statement which follows immediately after the macro instruction. If the macro instruction is in the source program, the next source program instruction statement is processed. If the macro instruction is a model statement in an outer macro definition, the next model statement is processed.

The sequence symbol .sym is used as a branch destination for AIF or AGO instructions.

Programming notes

1. The MEXIT instruction may only be used in macro definitions.
2. The MEXIT instruction may not be interchanged with the MEND instruction. The MEND instruction must always be the last instruction in a macro definition, even if this contains one or more MEXIT instructions.

Example

```
Name      | Operation | Operands
----------|-----------|-----------------------
          | MACRO     |
          | .         |
          | .         |
          | LCLB      | &OKIN1,&OKIN2
          | .         |
          | .         |
          | AIF       | (&OKIN1).OK1        (01)
          | MNOTE     | 'NO OKIN1'
          | MEXIT     |                     (02)
.OK1      | AIF       | (&OKIN2).OK2        (01)
          | MNOTE     | 'NO OKIN2'
          | MEXIT     |                     (02)
          | .         |
          | .         |
          | MEND      |
```

(01) Checks if a certain switch has been set, and branches to the next inquiry
(02) If OKIN is still on zero, the appropriate message is displayed, and generation is terminated.

MNOTE Transmit messages

Function

The MNOTE instruction generates a message (which can be changed using variable symbols) with a line number in the assembler listing.

Format

Name	Operation	Operands
[.sym]	MNOTE	[{*|n},]'text'

.sym	Sequence symbol
n	Error code, positive arithmetic macro expression; $0 \leq n \leq 255$
text	Text of message, maximum 256 characters

Description

Depending on the error code in the MNOTE instruction, warnings or error messages can be displayed in the assembler listing. An error message displayed as the result of an MNOTE is dealt with in the same way as an assembler flag, and the error weight is increased accordingly.

An MNOTE reference list can be created in the same way as as the error reference list (see "ASSEMBH User Guide" [1]).

* If an asterisk is given as the error code, then text appears as a remarks line in the generated code, even if the instruction PRINT NOGEN has been set. The message is purely a remark, and no error code is included. There is no display in MNOTE-XREF.

n The error code of the MNOTE instruction and the assembler flag error classes are related as follows:

warning	n = 0
error	1 ≤ n ≤ 150
severe error	151 ≤ n ≤ 254
termination error	n = 255

If n is omitted or if n > 255, the assembler assumes n = 0.
If n > 255, a warning is also displayed.

If n is not an integer, then the MNOTE instruction is regarded as an illegal model statement.

Programming notes

1. Variable symbols in the operand entry of the MNOTE instruction are replaced by their current value. Using MNOTE *,... remarks can be generated here.
2. An MNOTE instruction with the error code n = 255 can be used to abort assembly , if the appropriate termination condition has been set in the option specification (see "ASSEMBH User Guide" [1]).

MTRAC Macro trace

Function

The MTRAC instruction is used to monitor conditional branches and to check the contents of SET symbols.

Format

Name	Operation	Operands
[.sym]	MTRAC	

.sym Sequence symbol

Description

Macro instructions which follow after the MTRAC instruction are printed out in the assembler listing:

MTRAC, NTRAC, ANOP, LCLx and GBLx instructions
These instructions are logged with no additional details.

AIF and AGO instructions
With these two instructions, whether or not a branch was made is indicated by a "Y" or "N" in the assembler listing.

simple AIF	computed AIF	simple AGO	computed AGO
Y N	Y: <n> N	Y	Y: <n>

Y there was a branch
N there was no branch
n is the number of the logical expression that causes the branch to be made

SETA, SETB, SETC and ACTR instructions

Here, the current value of the SET symbol or ACTR operand is printed out in the assembler listing.

– SETA symbols and ACTR operands are shown as decimal numbers with leading zeros. Negative values in SETA symbols are printed with the preceding sign. Numbers with more than 8 positions are truncated to the right.

– The value of a SETB symbol appears as a single character:

  T ≙ true, or 1
  F ≙ false, or 0

– SETC symbols or symbolic parameters in SETC instructions are shown as a character string of up to 8 characters.

  The character string '␣␣␣␣NULL' stands for a SETC symbol which has been reset to the initial value.

  If the value of the SETC symbol is longer than 8 characters, only the first 7 characters and an asterisk (*) are printed.

Programming notes

1. If PRINT NOGEN was specified to control the contents of a listing, no MTRAC information is output.
2. If the MTRAC instruction is in the source program or in an outer macro, it must be executed in order to be valid for subsequent inner macro instructions.
3. The MTRAC attribute is passed down through the instruction hierarchy, in accordance with the macro nesting level (see example in NTRAC).

NTRAC Terminate macro trace

Function

The NTRAC instruction cancels the MTRAC function.

Format

Name	Operation	Operands
[.sym]	NTRAC	

.sym Sequence symbol

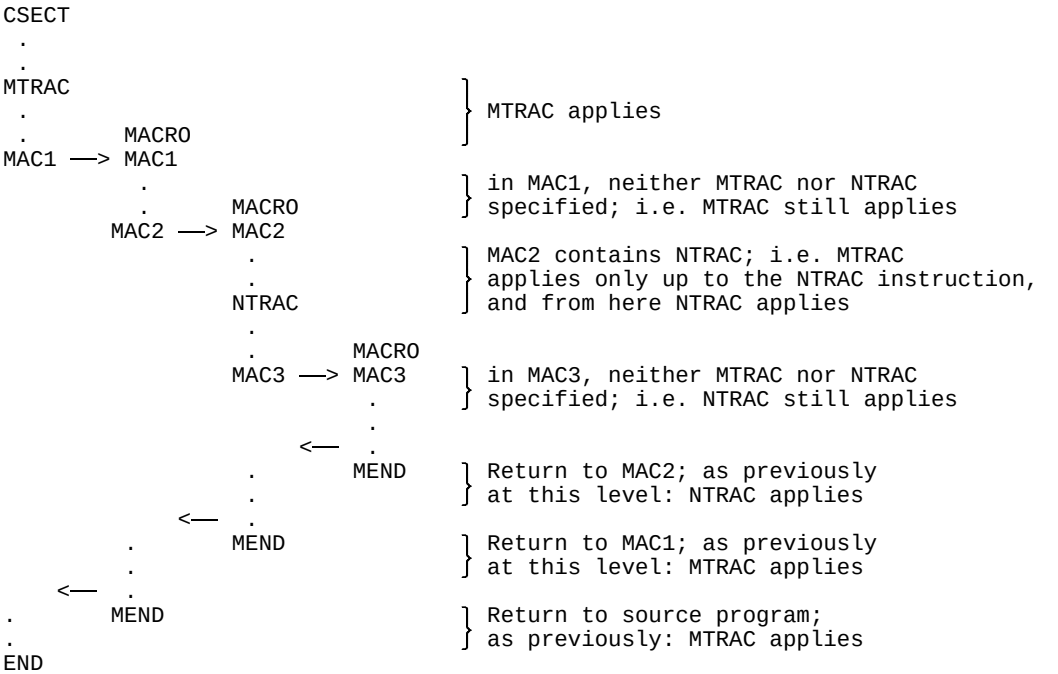
Programming notes

1. If no MTRAC instruction is specified, all NTRAC instructions processed are ineffective.
2. NTRAC resets the MTRAC instruction at the appropriate macro nesting level. NTRAC is valid for all subsequent inner macro instructions.

Example

The example shows how the MTRAC or NTRAC attribute is passed on to a source program which contains several nested macro instructions.

The source program calls the macro MAC1, which calls the macro MAC2, which in turn calls the macro MAC3.



SETA Set SETA symbol

Function

The SETA instruction assigns the arithmetic value from the operand entry to a SETA symbol.

Format

Name	Operation	Operands
[&par] {&par(d)}	SET[A]	[arexp] {&par_c}

&par SETA symbol
&par(d) Subscripted SETA symbol;
d Dimension; arithmetic macro expression
arexp Arithmetic macro expression
&par_c SETC symbol

Description

&par, or &par(d)

Local or global SETA symbol, which was defined in an LCLA or GBLA instruction.

If the SETA symbol was not defined before the SETA instruction, the assembler interprets it as an implicitly declared local SETA symbol and assigns the value arexp as the initial value. In this case, the mnemonic operation code SETA must be used.

Successive SETA instructions with the same SETA symbol in the name entry assign a new value to the symbol in each case.

arexp The value of the expression in the operand entry is calculated according to the rules for arithmetic macro expressions, and is then assigned to the SETA symbol.

This value is used instead of the SETA symbol if the symbol is used in an arithmetic expression.

&par_c A SETC symbol is only allowed as an operand entry if it has an arithmetic value.

An empty character string is converted into a 0.

Programming notes

If the SETA symbol is used in a non-arithmetic macro expression, the value of arexp is converted into an integer without leading zeros.

Example

```
Name        | Operation | Operands
------------------------------------------------
* Macro definition
            | MACRO     |
&NAME       | SETAEX    | &PART,&PARF
            | LCLA      | &P1,&P2,&P3,&P4
.*          |           |
&P1         | SETA      | 10                  (01)
&P2         | SETA      | 12                  (02)
&P3         | SETA      | &P1-&P2             (03)
&P4         | SETA      | &P1+&P3             (04)
.*          |           |
&NAME       | ST        | 2,SAVAREA
            | L         | 2,&PARF&P3          (05)
            | ST        | 2,&PART&P4          (06)
            | L         | 2,SAVAREA
            | MEND      |

* Macro call
BEGIN       | SETAEX    | FIELDX,FIELDY

* Generated instruction statements
            | .         |
            | .         |
BEGIN       | ST        | 2,SAVAREA
            | L         | 2,FIELDY2
            | ST        | 2,FIELDX8
            | L         | 2,SAVAREA
            | .         |
            | .         |
```

(01) and (02) assign the SETA symbols &P1 and &P2 the arithmetic values 10 and 12.

(03) and (04) utilization of the SETA symbol in arithmetic macro expressions; &P3 and &P4 contain the values -2, and +8, respectively.

(05) and (06) utilization of the SETA symbol in non-arithmetic macro expressions; thus, &P3 and &P4 are replaced here by 2 and 8, respectively.

SETB Set SETB symbol

Function

The SETB instruction assigns the logical value 1 or 0 (true or false) to a SETB symbol.

Format

Name	Operation	Operands
&par &par(d)	SET[B]	0 1 (0) (1) (logexp) &par_c

&par	SETB symbol
&par(d)	Subscripted SETB symbol
d	Dimension; arithmetic macro expression
logexp	Boolean expression
&par_c	SETC symbol

Description

&par or &par(d)

Local or global SETB symbol which was defined in an LCLB or GBLB instruction.

If the SETB symbol was not defined before the SETB instruction, the assembler interprets it as an implicitly declared local SETB symbol and assigns 0 or 1 as the initial value, depending on the value of the operand entry. In this case, the mnemonic operation code SETB must be used.

Successive SETB instructions with the same SETB symbol in the name entry assign a new value to the symbol in each case.

0, 1, (0) or (1)

These values are assigned to the SETB symbol and used in its place if the symbol is used in a Boolean expression.

logexp The logical value of the expression in the operand entry is calculated as per the rules for Boolean expressions, then assigned to the SETB symbol.

This value is used instead of the SETB symbol if the symbol is used in a Boolean expression.

&par_c A SETC symbol is only allowed as an operand entry if it has a value of 0 or 1.

An empty character string is converted to a 0.

Programming notes

If a SETB symbol is used in an arithmetic macro expression, the logical values 1 (true) or 0 (false) are converted to the corresponding arithmetic values +1 or +0.

Example

```
Name        | Operation | Operands
------------------------------------------------
* Macro definition
            | MACRO     |
&NAME       | SETBEX    | &P1,&P2
            | LCLB      | &B1,&B2
            | .         |
            | .         |
&B1         | SETB      | (L'&P1 EQ 4)        (01)
&B2         | SETB      | (S'&P2 EQ 0)        (02)
            | .         |
            | .         |
            | MEND      |

* Macro call
BEGIN       | SETBEX    | FIELDA,FIELDB

* Source program definitions
FIELDA      | DC        | F'01'
FIELDB      | DC        | DS3'12'
            | .         |
            | .         |
```

(01) The Boolean expression is true; &B1 is therefore assigned the value 1.
(02) The Boolean expression is false, hence &B2 is assigned the value 0.

SETC Set SETC symbol

Function

The SETC instruction assigns the value of the character expression in the operand entry to a SETC symbol or to a symbolic parameter.

Format

Name	Operation	Operands
[&par / &par(d) / &spar / &SYSLIST(n)]	SET[C]	charexp

&par	SETC symbol
&par(d)	Subscripted SETC symbol
d	Dimension; arithmetic macro expression
&spar	Symbolic parameter
&SYSLIST(n)	System variable symbol
charexp	Character expression

Description

&par or &par(d)

Local or global SETC symbol which was defined in an LCLC or GBLC instruction.

If the SETC symbol was not defined before the SETC instruction, the assembler interprets it as an implicit local SETC symbol and assigns the value of charexp as the initial value. In this case, the mnemonic operation code SETC must be used.

Successive SETC instructions with the same SETC symbol in the name entry assign a new value to the symbol each time.

&spar, or &SYSLIST(n)

Symbolic parameter to which the value in the operand entry is to be assigned. When &SYSLIST(n) is used, n may only denote the symbolic parameter itself. SETC cannot be used to assign a new value to an element in an operand sublist.

charexp

The value of the character expression is used instead of the SETC symbol if the symbol is used in an instruction statement.

Programming notes

When the SETC instruction is processed, the value of the operand is always interpreted as a string and not parsed into sublists. Thus, in &PAR SETC '(1,2)', for example, the value of &PAR(1) = (1,2), not 1.

Example

```
Name          | Operation | Operands
--------------------------------------------------
* Macro definition
              | MACRO     |
&NAME         | SETCEX    | &IN,&OUT
              | LCLC      | &PRE
*             |           |
&PRE          | SETC      | '&IN'(1,5)          (01)
*             |           |
&NAME         | ST        | 2,SAVAREA
              | L         | 2,&PRE&OUT          (02)
              | ST        | 2,&IN
              | L         | 2,SAVAREA
              | MEND      |

* Macro call
BEGIN         | SETCEX    | FIELDA,B

* Generated instruction statements
              | .         |
              | .         |
BEGIN         | ST        | 2,SAVAREA
              | L         | 2,FIELDB
              | ST        | 2,FIELDA
              | L         | 2,SAVAREA
              | .         |
              | .         |
```

(01) The SETC instruction assigns &PRE the value of the character substring (FIELD).
(02) &PRE is replaced by FIELD, and &OUT is replaced by B, i.e. &PRE&OUT receives the value FIELDB

8 Macro language elements in assembler source program text

Various macro language elements may be used **outside** macro definitions in the text of the assembler source program. The assembler source program can be regarded as a large macro definition in this case, except that it has no MACRO and MEND instructions or prototype statements.

The instruction statements generated using macro language are always shown in the assembler listing immediately after the corresponding macro instruction statements and are identified by a plus sign (+) in the macro level column.

The macro language elements which may be used in the assembler source program are detailed below.

- **Macro statements**

	Utilization outside macro definitions
ACTR	yes
AIF	yes
AGO	yes
ANOP	yes
GBLx	yes
LCLx	yes
MACRO	no
MEND	no
MEXIT	no
MNOTE	yes
MTRAC	yes
NTRAC	yes
SETA	yes
SETB	yes
SETC	yes

Programming notes

1. Instruction statements skipped by AIF or AGO are not read, and thus not assembled. Names defined in skipped instruction statements are regarded as undefined as far as the assembler is concerned. Their attributes can, however, be accessed by means of the lookahead function of ASSEMBH (see "ASSEMBH User Guide" [1]).
2. Instruction statements which are multiply executed a loop are also successively read in and assembled more than once.
3. If the value of the ACTR counter is exceeded (see section 7.2, ACTR instruction), assembly is abnormally terminated.

- **Variable symbols**

	Utilization Outside Macro Definitions
Symbolic parameters	no
SET symbols	yes if defined before first utilization *Exception:* implicitly declared local SET symbols need not be defined beforehand.
System var. symbols	
&SYSDATE	yes
&SYSECT	no
&SYSLIST	no
&SYSMOD	yes
&SYSNDX	no
&SYSPARM	yes
&SYSTEM	yes
&SYSTIME	yes
&SYSTSEC	no
&SYSVERM	no
&SYSVERS	yes

Programming notes

Variable symbols may also be used in the assembler source program to generate the name, operation, and operand entries of assembler instruction statements.

- **Sequence symbols**

	Utilization Outside Macro Definitions
Standard format	yes, the sequence symbol will be listed
Generated format	yes, in the operand entry

Programming notes

Sequence symbols in the operand entry of AIF and AGO instructions must also be defined in the name entry of an instruction statement in the source program.

- **Attribute references**

	Utilization outside Macro Definitions
Type attribute	yes, in macro statements
Length attribute	yes
Scaling attribute	yes, in macro statements
Integer attribute	yes, in macro statements
Count attribute	yes, in macro statements for SET symbols and system variable symbols
Number attribute	no
Definition attribute	yes, in macro statements

9 Structured programming with ASSEMBH

9.1 Introduction

Structured programming is not supported by ASSEMBH-BC !

Structured programming contains rules for the construction of programs. Its aim is to ensure that the control sequence flow of the program is clear and easily comprehensible.

Each program has a static and a dynamic aspect. The dynamic aspect is expressed in the written draft of the program, the source program text. The dynamic aspect consists of the series of actions which the program initiates in the computer. In the text of a source program are instruction statements, which determine the action which is to be carried out next. These instruction statements define the control flow of the program.

Structured programming assumes that the control flow of a program may consist only of certain basic formats. The following basic principles have been used as a foundation for this:

- **Block principle**

  The block principle requires that a program is formed only by stringing together or nesting structure blocks, which have only one entry and one exit. The control flow of a program may only be constructed from a few basic formats of this type with a simple inner structure.

- **Procedure principle**

  The control flow between main programs and subroutines is regulated in the procedure principle. A procedure in the sense of structured programming is a program unit with one entry and one exit, which can be called using a name, with current parameter values where necessary. A procedure is constructed from structure blocks.

- **Data principle**

  The data principle aids the user in making rational use of memory, and relieves him of the task of memory management which is required for reentrant procedures. The data principle regulates the scope of all data and ensures transparency. It requires that each procedure supply information on all data used.

ASSEMBH provides the user with aids to adapt assembler programs to the requirements of structured programming. This is achieved with the help of a set of predefined macros which can be used to implement the basic principles mentioned above.

In executing a program with predefined macros, ASSEMBH uses dynamic memory management. Accesses to and requests for memory, automatic register saving, and parameter management are handled by the ASSEMBH runtime system. The runtime system is implemented via a module package in the associated module library.

In addition to the set of macros and the runtime system, structured programming with ASSEMBH includes utilities for debugging and list editing.

This chapter describes the block, procedure and data principles in ASSEMBH, as well as the predefined macros and how to use them. The use of the utilities and the macro library, and how to link in the runtime system, is described in the "ASSEMBH (BS2000) User Guide" [1].

Using structured programming for program design

The basic principles of structured programming meet the requirements for a gradual, carefully developed procedure when designing programs.

The utilization of pseudocode lends itself to the formulation of flow logic. Pseudocode consists of formalized parts and of texts in natural language. Here, the predefined macros of ASSEMBH can be used for the formalized sections. The control flow of a program is formulated using predefined macros, and the function of each individual block is described in natural language. In the next stage, the function of the individual blocks can be worked out in detail, or individual blocks can be relocated as separate procedures.

Example

The example shows a decision (see section 9.2.2, "Selection structure blocks") in which only macros which represent structure blocks are currently specified.

```
@IF
condition
@THEN
operations, where condition applies
@ELSE
operations, where condition does not apply
@BEND
```

The ASSEMBH utilities convert the predefined macros into clearer forms of representation which can be output to a printer, and check the design for structural errors. As text between the macros is not processed by the utilities, the program design can be shown already structured with the aid of this pseudocode, without any assembly coding being present.

If an assembler program is to be worked out from the pseudocoded design, printing out the functions of the individual blocks in assembler instruction statements suffices. Here, we recommend leaving the text parts of the pseudocode as comments in the source program text. As before, the predefined macros create the control flow of the module implemented.

ASSEMBH converts the predefined macros into assembly language instruction statements and assembles them into the appropriate object module, along with the other instruction statements.

9.2 Block principle

A prerequisite of the block principle is that the program sequence and format of a procedure should be handled with only a few basic formats, the "structure blocks". A procedure may consist solely of structure blocks which are strung together or nested.

Every structure block has a simple control flow which defines the sequence of work stages contained in it. Basically, there are only three forms of control flow in a structure block:
- the sequence (several stages follow each other), section 9.2.1
- the selection (one of several possible stages is executed), section 9.2.2
- the loop (one stage is repeated several times), section 9.2.3.

As a result, the control flow in a structure block has only one entry and one exit.

Several structure blocks may be strung together and produce another sequence as a result, a parent structure block. Structure blocks may also be nested. In other words, one stage in a sequence, one alternative in a selection structure block, or the repeated section of a loop may itself consist of a structure block.

In the structure blocks for decision, loop with pre-check, loop with unqualified terminal condition, and count loop with unqualified terminal condition, a condition must be specified for the macro calls @IF, @WHIL and @WHEN. There are:
- simple conditions, section 9.2.4, and
- compound conditions, section 9.2.5.

For graphic representation of structure blocks, Nassi-Shneiderman diagrams (structograms) are generally used. ASSEMBH provides the user with a set of predefined macros with which every structogram can be coded on a 1:1 basis.

All branches in a program are implemented using these macros, hence branch instructions (B, BAL, BALR, ...) should basically not be used.

9.2.1 Sequence

The sequence structure block combines program sections into a logical unit. It can be used anywhere, and its chief purpose is clear organization, while facilitating the assignment of names to strings of instruction statements and structure blocks.

The sequence structure block is mandatory for delimiting sub-blocks in case differentiation by number (@CASE).

Nassi-Shneiderman diagram

Structure block format

Name	Operation	Operands
[name]	@BEGI	
	{ [instr. statement] / [structure block] } [,...]	
[name]	@BEND	

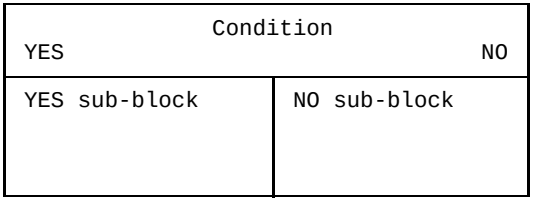
9.2.2 Selection structure blocks

A selection structure block consists of a series of sub-blocks and a criterion which selects only one of the sub-blocks.

9.2.2.1 Decision

The decision involves choosing between two sub-blocks. The selection criterion is a condition which is composed of a condition symbol and a machine instruction which sets the condition code (see section 9.2.2, "Conditions").

Nassi-Shneiderman diagram



Structure block format

Name	Operation	Operands
[name]	@IF	cond_sym [condition code setting machine instruction]
[name]	@THEN	YES sub-block
[[name]	@ELSE	NO sub-block]
[name]	@BEND	

cond_sym predefined or user-own condition symbol; specifies on what criterion the condition may be interrogated.

sub-block string of instruction statements or other nested structure blocks.

Description

The YES sub-block is initiated by the @THEN macro, and the NO sub-block by @ELSE. @BEND ends the structure block.

If the condition (see sections 9.2.4 and 9.2.5) is true, the instruction statements or structure blocks in the YES sub-block are executed; if not, the NO sub-block is. If the NO sub-block is missing, the instruction statement following the @BEND macro is executed as a result of a non-applicable condition.

Example

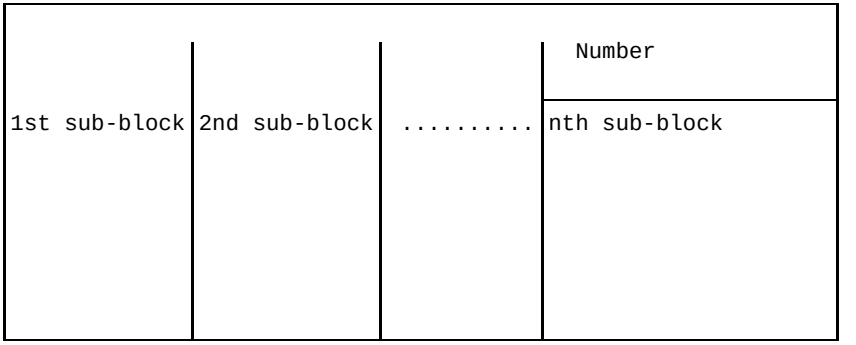
```
Name        | Operation | Operands
------------+-----------+-----------------
            | @IF       | LE
            | CR        | R1,R2
            | @THEN     |
            | MVI       | FIELD,TRUE
            | @ELSE     |
            | MVI       | FIELD,FALSE
            | @BEND     |
```

If the contents of register 1 are less than/equal to those of register 2 the YES sub-block is executed; if not, the NO sub-block is.

9.2.2.2 Case differentiation by number

In case differentiation by number, one of several sub-blocks is selected. The selection criterion is the number of the corresponding sub-block. This number is specified in the @CASE register.

Nassi-Shneiderman diagram



Structure block format

Name	Operation	Operands
[name]	@CASE	(reg)
[name]	@BEGI	
	1st sub-block	
[name]	@BEND	
	.	
	.	
	.	
[name]	@BEGI	
	nth sub-block	
[name]	@BEND	
[name]	@BEND	

reg @CASE register; contains the number of the sub-block to be executed
sub-block String of instruction statements or other nested structure blocks

Description

The @CASE register must be loaded with the number of the appropriate sub-block prior to execution of the structure block. A maximum of 90 sub-blocks are possible for the structure block.

Register 0 may not be used as @CASE register.

The contents of the @CASE register are altered during execution of the @CASE macro.

If the contents of the @CASE register is less than 1 or greater than the number of sub-blocks defined,
- the last sub-block is executed if CHECK=ON is set in the procedure heading (see section 10.2, @ENTR);
- program errors occur if CHECK=OFF is set in the procedure heading.

Sub-blocks in a case differentiation by number must be delimited with a sequence (@BEGI, @BEND).

Example

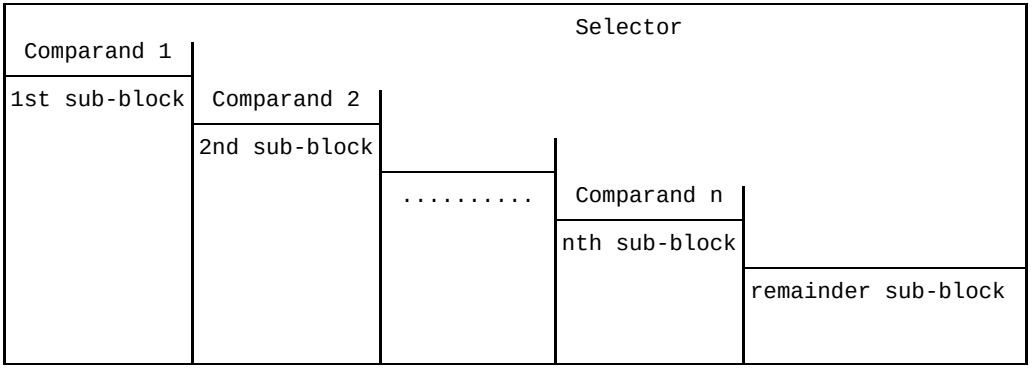
```
Name        | Operation | Operands
------------+-----------+-----------------------------
            | L         | R6,T2
            | @CASE     | (R6)
            | @BEGI     |          ┐
            | .         |          } 1st sub-block
            | .         |          ┘
            | @BEND     |
            | @BEGI     |          ┐
            | .         |          } 2nd sub-block
            | .         |          ┘
            | @BEND     |
            | .         |          ┐ other sub-blocks
            | .         |          ┘ possible
            | @BEND     |
            | .         |
            | .         |
T1          | DC        | F'1'
T2          | DC        | F'2'
            | .         |
            | .         |
```

In the example, register 6 is used for the @CASE branch. The content of register 6 is 2 in this case; hence the second sub-block is executed.

9.2.2.3 Case differentiation by comparison

In case differentiation by comparison, one of several sub-blocks is selected. This is done based on a comparison of selector and comparands. The selector is specified in the structure block heading, and the comparands in the sub-block heading.

Nassi-Shneiderman diagram



Structure block format

Name	Operation	Operands
[name]	@CAS2	⌈name ⌉ {literal}[,COMP=instr] ⌊(reg) ⌋
[name]	@OF	⌈name ⌉ {literal}[,...][,COMP=instr] ⌊val ⌋
	1st sub-block	
	.	
	.	
	.	
[[name]	@OFRE	
	rem. sub-block]	
[name]	@BEND	

name or literal or (reg) in the operand entry of @CAS2
: specifies the name of the field or register which contains the selector or, in the form of a literal, the selector itself.

name or literal or val in the operand entry of a @OF
: specifies the name of the field which contains the appropriate comparands, or the comparand itself in the form of a literal or self-defining term.

sub-block
: String of instruction statements or other nested structure blocks

instr
: Condition code setting machine instruction

Description

The current value of the selector in the @CAS2 macro is compared with the comparands in the @OF macro. The comparison is done in the given sequence of comparands, from top to bottom, and from right to left. On finding the first selector and comparand that match, the relevant sub-block is processed, and the structure block is then quit.

If there is no matching comparand, the remainder sub-block is processed. If this is missing, the structure block is quit, and the instruction statement after the @BEND macro is processed.

The end of a sub-block is indicated by the next @OF, @OFRE or @BEND macro which terminates the sub-block.

Comparison is conventionally carried out with the CLC instruction, and in registers with the C instruction. A different type of comparison may be selected using the COMP= operand. Here, the operand in @CAS2 is valid for the entire structure block, but in @OF macros only for the sub-block concerned.

The user must himself ensure that the generated machine instructions

```
compare instruction     selector,comparand
```

are correct, i.e. that implied or explicit length specifications and operand alignment are correct.

In compare instructions, the sub-block is executed if there is equality (EQ); in COMP=TM, if all the bits - corresponding to the mask - in the selector are set (ON); in COMP=TRT, if a function byte unequal to zero (NZ) has been found in the comparand (conversion table).

There is no restriction on the number of sub-blocks in the structure block and the number of comparands in an @OF.

To enable selection of the correct sub-block, all comparisons preceding the searched for sub-block will be carried out. It is therefore advisable to organize sub-blocks and comparands according to expected frequency of occurrence.

Example

```
Name      | Operation | Operands
----------+-----------+-------------------------
BLOC      | @CAS2     | KEY,COMP=CLI          (01)
          | @OF       | 5                     (02)
          | .         |
          | .         |
          | @OF       | 6,8                   (03)
          | .         |
          | .         |
          | @OF       | END,COMP=CLC          (04)
          | .         |
          | .         |
          | @OF       | X'80',COMP=TM         (05)
          | .         |
          | .         |
          | @BEND     |
```

(01) The selector is in the field KEY; the compare instruction for the structure block is CLI.

(02) Comparand for the first sub-block is the self-defining term 5, i.e. the first sub-block is executed if the current value of KEY is equal to 5.

(03) The self-defining terms 6 and 8 are specified as comparands for the second sub-block. In other words, it is executed if the value of KEY is equal to 6 or 8.

(04) Alteration to the compare instruction for this sub-block; it is processed if the value of KEY is equal to the value of END.

(05) For the last sub-block, another compare instruction is specified. It is executed if the bits - corresponding to the mask - in the selector are all set.

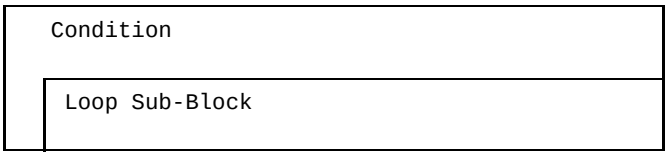
9.2.3 Loops

A loop consists of a loop sub-block and a condition which specifies how often and/or how long this is to be repeated.

9.2.3.1 Loop with pre-check

In a loop with pre-check, the condition is always checked prior to execution of the loop sub-block.

Nassi-Shneiderman diagram



Structure block format

Name	Operation	Operands
[name]	@WHIL	cond_sym [condition code setting machine instruction]
[name]	@DO	 loop sub-block
[name]	@BEND	

cond_sym Predefined or user-own condition symbol; specifies on what criterion the condition is to be interrogated.

loop sub-block
String of instruction statements or other nested structure blocks.

Description

The loop sub-block is executed if the condition (see section 9.2.2) is satisfied. If the condition is not satisfied, the loop is quit before the loop sub-block.

If the condition is not satisfied in the first check, the loop sub-block is not executed.

Example

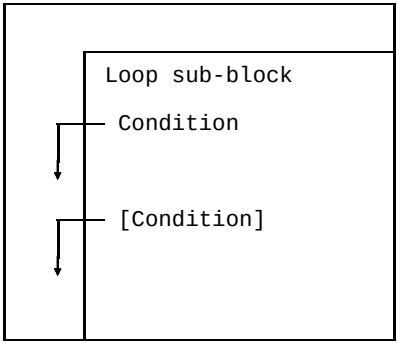
```
Name        | Operation | Operands
------------|-----------|-----------------
LOOP        | @WHIL     | EQ
            | CLI       | END,C'N'
            | @DO       |
            | .         |
            | .         |
            | @BEND     |
```

The loop sub-block is executed provided END is equal to "N".

9.2.3.2 Loop with unqualified terminal condition

In a loop with unqualified terminal condition, one or more terminal conditions may be specified at any desired points in the loop sub-block.

Nassi-Shneiderman diagram



Structure block format

Name	Operation	Operands
[name]	@CYCL	
	.	
	.	
	.	
[name]	@WHEN	cond_sym
	[condition code setting machine instruction]	
[name]	@BREA	
	.	
	.	
	.	
[name]	@BEND	

cond_sym Predefined or user-own condition symbol; specifies on what criterion the condition is to be interrogated.

Description

The loop sub-block is limited by @CYCL and @BEND. It is repeated so long as one of the terminal conditions prevail. Then the loop is quit at the specified position with @BREA.

The conditions set (see sections 9.2.4 and 9.2.5) are always checked when they occur.

@WHEN-@BREA can be used more than once in the loop sub-block.

Additional structure blocks may be nested in the loop sub-block. These may only be quit with @WHEN-@BREA if they are also @CYCL loops. In @CYCL loops nested in this way, a @WHEN-@BREA condition only terminates the loop in which it is defined, not the entire structure block.

Example

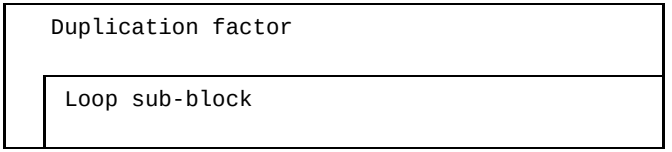
Register 3 contains the address of a character string. This is to be searched for the position of the field separator '*'.

```
Name        | Operation | Operands
------------|-----------|-----------------
            | @CYCL     |
            | @WHEN     | EQ
            | CLI       | 0(R3),C'*'
            | @BREA     |
            | AH        | R3,ONE
            | @BEND     |
            | .         |
            | .         |
ONE         | DC        | H'1'
```

9.2.3.3 Count loop

In a count loop, the sub-block is repeated as often as specified in the duplication factor.

Nassi-Shneiderman diagram



Structure block format

Name	Operation	Operands
[name]	@CYCL loop sub-block	(reg)
[name]	@BEND	

reg Register whose contents specify the number of iterations

Description

reg is decremented by 1 after each execution, and checked for 0. If 0 is reached, the structure block is quit.

If the contents of reg are already less than 1 at the start,
- the loop is terminated immediately if CHECK=ON is set in the procedure heading (see section 10.2, @ENTR),
- a continuous loop ensues if CHECK=OFF is set, and the loop has no terminal condition.

reg may not be altered in the loop sub-block.

Example

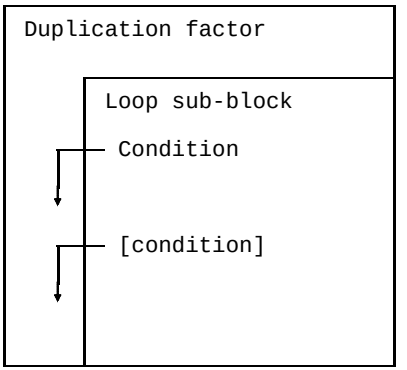
Table 2 (TAB2) is to be copied into Table 1 (TAB1). The number of elements is specified in Field N, the length of each element in L.

```
Name     | Operation | Operands
---------+-----------+-----------------
         | L         | R5,N
         | LA        | R6,TAB1-L
         | LA        | R7,TAB2-L
LOOP     | @CYCL     | (R5)
         | LA        | R6,L(0,R6)
         | LA        | R7,L(0,R7)
         | MVC       | 0(L,R6),0(R7)
         | @BEND     |
```

9.2.3.4 Count loop with unqualified terminal condition

The count loop with unqualified terminal condition combines the characteristics of the count loop with those of the loop with unqualified terminal condition. At the same time, there are @WHEN-@BREA terminal conditions and a duplication factor. The duplication factor forms an upper limit for the number of executions of the loop sub-block.

Nassi-Shneiderman diagram



Structure block format

Name	Operation	Operands
[name]	@CYCL	(reg)
	.	
	.	
	.	
[name]	@WHEN	cond_sym
	[condition code setting machine instruction]	
[name]	@BREA	
	.	
	.	
	.	
[name]	@BEND	

reg Register whose contents specifies the number of iterations

cond_sym Predefined or user-own condition symbol; specifies on what criterion the condition is to be interrogated.

Description

The loop sub-block is delimited by @CYCL and @BEND. It is repeated as often as specified by the duplication factor, or as long as one of the terminal conditions prevail. The loop is then quit at the specified position with @BREA.

The conditions set are always checked when they occur.

@WHEN-@BREA may be set more than once in a loop sub-block.

Additional structure blocks may be nested in the loop sub-block. These may only be quit with @WHEN-@BREA if they are also @CYCL loops. In @CYCL loops nested in this way, a @WHEN-@BREA stipulation only ends the loop in which it is defined, not the entire structure block.

reg is decremented by 1 after each execution, and checked for 0. If 0 is reached, the structure block is quit.

reg may not be altered in the loop sub-block.

Example

The loop in the example is executed ten times, unless the termination condition occurs beforehand.

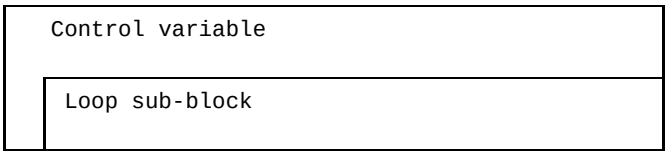
```
Name        | Operation | Operands
------------------------------------------------
            | L         | R7,NR                (01)
LOOP        | @CYCL     | (R7)                 (02)
            | .         |
            | .         |
            | @WHEN     | EQ                 ⎫
            | CLC       | TEST,END           ⎬ (03)
            | @BREA     |                    ⎭
            | .         |
            | .         |
            | @BEND     |
            | .         |
            | .         |
NR          | DC        | F'10'
TEST        | .         | .
END         | .         | .
```

(01) Register 7 is loaded with the duplication factor
(02) Start of loop
(03) Termination condition and loop exit

9.2.3.5 Iterative loop

In an iterative loop, the initial and end values of the control variable determine the number of iterations of the loop sub-block.

Nassi-Shneiderman diagram



Structure block format

Name	Operation	Operands
[name]	@THRU	(reg1),{(reg2) / name / literal}[,{(reg3) / name / literal}]
[name]	@DO loop sub-block	
[name]	@BEND	

(reg1) Register; specifies the initial value of the control variable

(reg2) or name or literal in the second operand
Register, name of a constant or literal which specifies the end value of the control variable. The constants or literal must be aligned on a halfword for this, and must be a halfword long.

(reg3) or name or literal in the third operand
Register, name of a constant or literal which specifies the increment. The constant or literal must be aligned on a halfword here, and be a halfword long.

Description

The first operand of @THRU (reg1) must be loaded with the initial value of the control variable prior to the call.

Whether the sum of control variable and increment exceeds the end value is queried prior to each execution of the sub-block. If this is the case, the structure block is quit.

After each execution of the loop sub-block, the increment is added to the initial value or actual value of the control variable. The actual value of the control variable is always in reg1.

The loop sub-block is repeated as often as specified by the integer part of the following expression:

```
((end value-initial value)/increment)+1
```

If no increment is specified, register 0 may not be used for the initial value.

An @THRU macro with the initial value ≤ end value and increment ≤ 0 results in a continuous loop.

Example

```
Name        | Operation | Operands
------------|-----------|-----------------
            | .         |
            | .         |
            | LH        | R6,BEGL
LOOP        | @THRU     | (R6),ENDL,PLUS
            | @DO       |
            | .         |
            | .         |
            | @BEND     |
            | .         |
            | .         |
BEGL        | DC        | H'1'
ENDL        | DC        | H'10'
PLUS        | DC        | H'2'
```

In this example, the initial value of the control variable is 1, the end value 10, and the increment 2. The loop sub-block is therefore executed 5 times.

9.2.4 Simple conditions

Format

Name	Operation	Operands
		cond_sym [condition code setting machine instruction]

cond_sym Predefined or user-own condition symbol (see below); specifies on what criterion the condition is to be interrogated

condition code setting machine instruction
Instruction executed immediately before the interrogation. The instruction which sets the interrogating condition code need not follow immediately after the condition symbol. If it is outside the condition, the condition code may not be altered by another instruction between the setting of the condition code and the interrogation. Condition code interrogation is done using @THEN, @DO and @BREA.

9.2.4.1 Predefined condition symbols

In structured programming there are a number of predefined condition symbols. In the table below, they are classified according to the most frequent compare results.

Results of compare instructions

=	EQ	equal
<	LT	less than
>	GT	greater than
	NE	not equal
≥	GE	greater or equal
≤	LE	less or equal

Characteristics of results of arithmetic and Boolean operations

=0	ZE	zero
<0	LZ	less than zero
>0	GZ	greater than zero
0	NZ	not zero
overflow	ON	

Results of the TM instruction (test under mask)

binary ones only	ON	ones
binary zeros only	ZE	zeros
ones and zeros mixed	MI	mixed
zeros only or ones only	ZO	ZE or ON
at least one binary zero	ZM	ZE or MI
at least one binary one	OM	ON or MI

Table 9-1 Classification of predefined condition symbols according to the most frequent instruction results

Examples

```
Name        | Operation | Operands
------------|-----------|---------
            | @...      | LZ
            | LTR       | R1,R1
```

The condition is true if the contents of register R1 are negative (less than zero). The instruction LTR R1,R1 has no effect other than setting the condition code.

```
            | @...      | GZ
            | S         | R1,ALPHA
```

The condition is true if the result of the subtraction register R1 minus ALPHA is greater than zero, otherwise it is false. In addition to setting the condition code, the register contents are also altered.

```
            | @...      | ON
            | TM        | 0(RPEGEL),MASKE
```

The condition is true if all the bits (selected with the MASKE mask) of the byte with the address in register RPEGEL are binary ones.

```
            | @...      | EQ
            | CLC       | KANDIDAT,0(RLISTE)
```

The condition is true if the contents of KANDIDAT are equal to the value of the same length, which is addressed via the register RLISTE .

9.2.4.2 User-own condition symbols

Over and above the predefined condition symbols, the user can assign individual condition symbols to the available condition masks. These must always begin with an '@'.

Allocation format

Name	Operation	Operands
@name	EQU	mask

The format of the mask in the operand entry of the EQU instruction can be seen in the following table.

(Condition symbols in the same line have the same value).

Condition code				Mask for the conditional branch		Corresponds to the predefined condition symbol
0	1	2	3	binary	numeric	
set				1000	8	EQ ZE
	set			0100	4	LT LZ MI
		set		0010	2	GT GZ
			set	0001	1	ON
not set				0111	7	NE NZ OM
	not set			1011	11	GE ZO
		not set		1101	13	LE
			not set	1110	14	ZM

Table 9-2 Masks for allocation of user-own condition symbols

9.2.5 Compound conditions

A compound condition consists of two or more simple conditions, which must be linked using the macros @TOR, @AND and @OR.

Format

Name	Operation	Operands
[name]	{@IF / @WHEN / @WHIL}	cond_sym
	condition code setting machine instruction	
[name]	{@TOR / @AND / @OR}	cond_sym
	condition code setting machine instruction	

cond_sym Predefined or user-own condition symbol (see section 9.2.4)

condition code setting machine instruction
Instruction that is executed immediately before the condition code interrogation.

In a compound condition, each simple condition must contain the condition code setting machine instruction.

In some cases, not all condition code setting instructions are executed. If, for example, the first condition of three simple conditions in an AND operation has been detected as not satisfied, the two remaining conditions, including the instructions contained in them, are ignored. We therefore recommend that, in compound conditions, only those instructions whose only effect is to set the condition code be used.

In a compound condition,
- @TOR implements "OR with priority",
- @AND implements "AND", and
- @OR implements "OR".

The linkage priority of macros which represent logical operators corresponds to the sequence listed. @TOR , in the same way as @OR, implements the inclusive-OR operation, but has higher linkage priority. In other words, by using @TOR instead of @OR, the processing sequence is altered with regard to @AND.

Examples

```
Name        | Operation | Operands
------------+-----------+-------------------
            | @CYCL     |
            | @WHEN     | EQ       Condition 1
            | CR        | R1,R2
            | @AND      | EQ       Condition 2
            | C         | R3,N
            | @TOR      | EQ       Condition 3
            | C         | R4,M
            | @BREA     |
            | .         |
            | .         |
```

The loop is quit if one of the two cases is given:
– Condition 1 and Condition 2 apply
– Condition 1 and Condition 3 apply.

```
Name        | Operation | Operands
------------+-----------+-------------------
            | @CYCL     |
            | @WHEN     | EQ       Condition 1
            | CR        | R1,R2
            | @AND      | EQ       Condition 2
            | C         | R3,N
            | @OR       | EQ       Condition 3
            | C         | R4,M
            | @BREA     |
            | .         |
            | .         |
```

The second example is the same as the first, except for the use of @OR in place of @TOR in condition 3.

The loop is quit here if one of these two cases apply:
– Condition 1 and Condition 2 prevail
– Condition 3 only prevails.

9.3 Procedure principle

The control flow between main programs and subroutines is regulated in the procedure principle. A procedure in the sense of structured programming is a program unit with one entry and one exit, which can be called using a name, if necessary with current parameter values. For a procedure text - unlike structure blocks - the dynamic and static end may be different.

A procedure is created from structure blocks strung together or nested. If a structure block gets too complicated or too extensive, it can be relocated and described in a separate procedure.

A procedure can be assembled into a separate object module. There may also be two or more procedures in an object module, however.

One procedure may call another. During the call, the calling procedure can pass parameters to the called procedure. On return to the calling procedure, the called procedure can issue a return value. The calling procedure is continued with the instruction statement which follows directly after the call.

Procedures cannot be statically nested inside each other.

Structured programming requires that programs be created from reentrant procedures.

A procedure is termed "reentrant" if several application programs run simultaneously through a procedure code (which is present only once). If this is the case, one program occupies the processor, while the others are in a wait state; their register states are temporarily stored in a register save area.

Data areas which are part of the code of a reentrant procedure may thus contain only constants; at runtime, data areas for variables must be requested and managed separately.

In structured programming, a distinction is made between
- data areas which are made ready when loading the program and which are retained for the entire program run (static areas),
- areas which are requested during the program run and released (controlled areas) and
- areas which are retained during a procedure run (automatic areas).

If the generated procedures are reentrant, statically prepared data areas may only be used for storing constants.

A definition must be contained in the procedure for each data area accessed in a procedure; the definition gives information regarding the storage class and type of access.

9.3.1 Procedure declaration and procedure end

A procedure is statically delimited by the macros @ENTR (procedure start) and @END (static procedure end). During the program run, a procedure is called with @PASS and terminated with @EXIT (dynamic procedure end). The following figure shows the dynamic linkage of several procedures via the @PASS call and the return to the calling procedure with @EXIT.

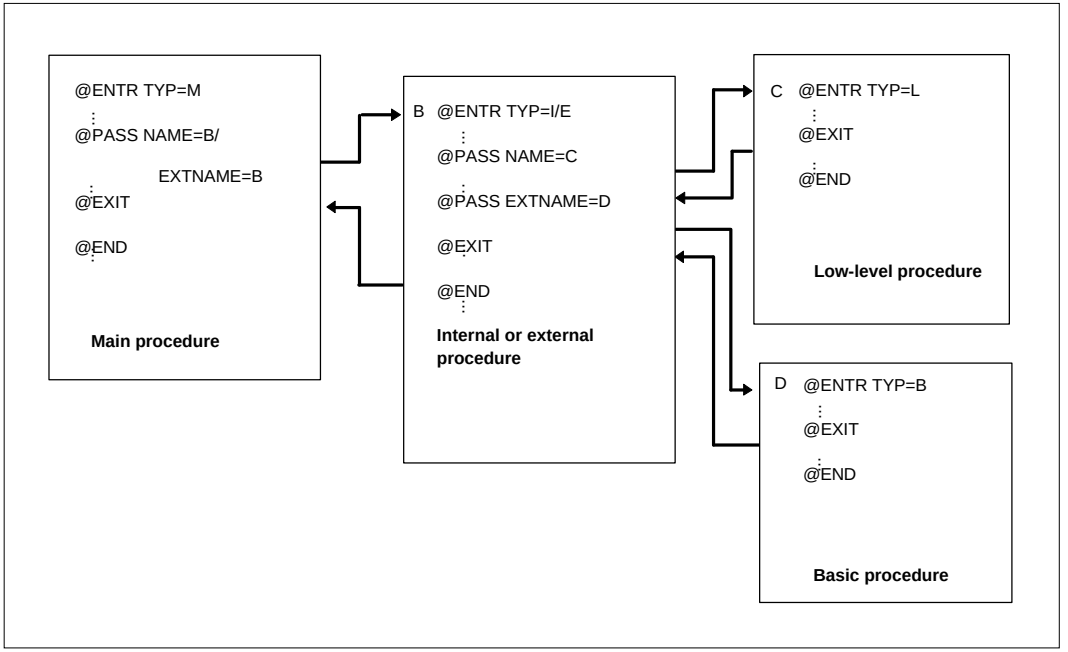


Fig. 9-1 Dynamic procedure linkag

Procedure declaration

The @ENTR macro represents the heading for all types of procedure. Here, the type of procedure, as well as its name, must be specified.

Over and above the main procedure, called from the operating system using the /START-PROGRAM or /LOAD-PROGRAM (Type M), external and internal procedures are differentiated. A procedure is termed "external" if it can be called from any module, and "internal" if it can be called within a module only.

External procedures may be of types E, B, D; internal procedures may be of types I, L, and D.

The @ENTR call also gives information on whether the procedure accepts parameters from another procedure, and/or itself passes parameters to another procedure. The number of parameters and form of passing is also indicated, as well as the data areas which must be made ready for this (see sections 9.5 and 10.2, @ENTR).

Procedure end

Here, the static and dynamic procedure end must be differentiated.

The static procedure end is implemented by the @END macro. This macro releases all base and addressing registers which were assigned for this procedure by ASSEMBH.

The @EXIT macro identifies the dynamic procedure end. In a main procedure, the program is terminated using @EXIT.

In a called procedure, control is returned to the calling procedure, which is continued with the instruction statement that follows the procedure call. A return value may be transferred to the calling procedure.

The return value is calculated in the called procedure. Its address is is either specified, or the return value transferred directly in the form of a self-defining term in the @EXIT macro (see section 10.2, @EXIT). In the calling procedure, this value may then be further processed or evaluated.

9.3.2 Procedure types

9.3.2.1 Type M, E and I procedures

Type M, E and I procedures are connected to memory management via the runtime system. When they are opened, the register is automatically saved, and reloaded when they are terminated. Memory areas for saving registers and for data required at runtime are readied dynamically.

In addition to class S and B (static and based) data areas, class A and C (automatic and controlled) data areas may also be requested (see section 9.4, "Data principle").

The base address register allocated for all three types is register 10. It may also be accessed using R@BASE.

Type M procedures

A type M procedure is the main procedure in a program. Program execution begins and ends in it. Via the @ENTR macro, a CSECT instruction is generated with the name of the program. Even when procedures from several modules are to be combined into one program, there may be only one type M procedure.

A type M procedure may not be called by another. It may therefore not accept parameters from another procedure.

Passing of parameters from the main procedure to other procedures may be static or dynamic, via the STANDARD or OPTIMAL interface (see section 9.5, "Procedure linkage and parameter passing").

Type E and I procedures

A type E procedure may be called from any module (external procedure)

A type I procedure can only be called within the module in which it lies (internal procedure).

Procedures of both types can also call internal or external procedures. They may pass parameters to the called procedure and also accept parameters themselves from a calling procedure. Passing of parameters is both static and dynamic, via the STANDARD or OPTIMAL interface; acceptance of parameters can only be done statically (see section 9.5, "Procedure linkage and parameter passing").

On return to the calling procedure, the registers allocated by ASSEMBH are reloaded with the original values. This facility can be extended or limited by means of the operand RETURNS= in the @ENTR macro (see @ENTR, format 2).

In E procedures, one CSECT instruction with the procedure name is generated as standard by the @ENTR macro. Also one ENTRY instruction only may be generated, parameter-controlled, instead of the CSECT instruction.

In I procedures, no CSECT instruction is generated. In other words, they belong to the preceding control section. The user must take care that an I procedure is not in the area of a dummy section (DSECT instruction).

Example

```
Name      | Operation | Operands
----------+-----------+-------------------
MAIN      | @ENTR     | TYP=M
          | @DATA     | ...,DSECT=DUMMY,...
          | .         |
          | .         |
          | @EXIT     |
          | @END      |
DUMMY     | DSECT     |
          | .         |
          | .         |
LDUMMY    | EQU       | *-DUMMY
MAIN      | CSECT     |
INTERN    | @ENTR     | TYP=I
          | .         |
          | .         |
```

9.3.2.2 Type B, L and D procedures

Type B, L and D procedures are not connected by the runtime system to memory management. No memory areas are reserved for register saving or dynamic parameter passing. On opening of these procedures, registers are not saved, nor are they reloaded on termination of the procedures.

Types B, L and D are provided for procedures which are on a low level in the call hierarchy, and accordingly execute basic functions.

In these procedures, the user himself must take care of memory management. No memory requests (class A and C data areas, see section 9.4) may be made.

Type B and L procedures

A type B (basic procedure) can be called from any module.

A type L procedure (low-level procedure) can only be called within the module in which it stands.

Register 15 is assigned as the standard base address register for both types. The user can assign another register as the base address register in the BASE operand of the @ENTR macro. This register must be loaded explicitly at the start of the procedure with the correct value (e.g. with the machine instruction "LR reg,R15").

Loading of the base address register after each call of another subprocedure is controlled via the operand LOADSB= in the @ENTR macro. LOADSB=YES only needs to be specified if the procedure has an alternative base register and calls another procedure.

Type B and L procedures may in turn call other external or internal procedures. Only static passing of parameters to the called procedure is possible.

In type B procedures, a CSECT instruction with the procedure name is generated as standard by the @ENTR macro. An ENTRY instruction may be generated, parameter-controlled, instead of the CSECT instruction.

In type L procedures, no CSECT instruction is generated. In other words, they belong to the preceding control section. The user must ensure that an L procedure does not lie in the area of a dummy section (DSECT instruction).

Type D procedures

Type D procedures (dummy procedures) are meant for simple operations which can be done with little effort. For this type, the loading of the base register must also be carried out explicitly by the user. D procedures may be both internal and external.

The following are **not** allowed in a type D procedure:
- calling of other sub-procedures using @PASS,
- return to the calling procedure with @EXIT,
- memory request and memory release.

If the first procedure is a type D module, no procedure name may be specified. The procedure receives the name of the START or CSECT instruction.

9.4 Data principle

The data principle of ASSEMBH assists the user in making rational use of memory, and relieves him of the task of memory management required for the creation of reentrant procedures. The data principle regulates the scope of definition of all data and ensures transparency. It requires that each procedure provide information on all data used. In a procedure, what data is used, where the data is, and how access is organized must be specified.

The following holds true for reentrant procedures:
- data areas which are part of a procedure may contain only constants.
- data areas for variables must be requested at runtime.

Data areas which are reserved on loading of the program and are available during the entire program run are accordingly differentiated from areas which are requested and released during the program run.

With dynamically requested areas, a distinction is made between those which are automatically released on quitting the procedure in which the request was made, and those which are controlled and released via an explicit instruction statement.

The memory areas of a program can be subdivided into the following four classes, in accordance with these distinctions:
- static,
- automatic,
- controlled and
- based.

Allocation of a class to an area is defined by the user in the @DATA macro. The @DATA macro is used to implement access for static and based areas, to request storage space and to implement access for automatic and controlled areas.

Areas of the static class are assigned by "@DATA CLASS=S" (see section 10.2, format 2 of @DATA). During loading of the program, they are allocated storage space which is retained during the entire program run. The @DATA macro loads a base address register for that purpose.

For areas of the automatic class, memory is reserved with "@DATA CLASS=A" (see section 10.2, format 1 of @DATA) or via a LOCAL area request (section 9.4.2). The memory area is reserved within the STACK area of the procedure. It is automatically released when the procedure in which it was requested is quit using @EXIT.

For areas of the controlled class, "@DATA CLASS=C" (see section 10.2, format 2 of @DATA) is used to reserve an area in the HEAP memory. It is then only released if the user expressly requests it via the @FREE macro. This may also be done in another procedure.

If an area which was assigned one of the classes static, automatic or controlled in a procedure is accessed in a lower-level procedure or one which is to be executed at a later stage, it must be identified there by "@DATA CLASS=B" (see section 10.2, format 3 of @DATA) and a register specified which contains the address of the area. This results in it being assigned the class attribute "based" for this procedure.

Registers 0, 1, 13, 14, 15 and the procedure base register may not be used as base address register for a data area defined using @DATA.

9.4.1 Data areas of the static class

Data areas of the static class are meant for constant data. Access to static areas is implemented with "@DATA CLASS=S". These areas are defined statically in the program, with the result that on loading the program, storage space - which is retained until the end of the program - has already been reserved. Data in a static area is available throughout the entire program run.

Storage space for static areas must be reserved either
- internally after the static end of the procedure (after @END and before @ENTR, or the END instruction) or
- externally in a separate data module.

The register specified in the BASE operand of @DATA is assigned as the base address register.

Internal static areas

The INIT operand of @DATA specifies the symbolic address of the data area.

Example

```
Name      | Operation | Operands
----------|-----------|-----------------------------
EX1       | @ENTR     | TYP=E
          | .         |
          | .         |
          | @DATA     | CLASS=S,BASE=R5,INIT=CONST
          | .         |
          | .         |
          | @EXIT     |
          | @END      |
CONST     | DS        | 0CL180
C1        | DC        | C'ABC'
          | .         |
          | .         |
```

Register 5 is allocated as the base address register for the static area in the EX1 procedure. This area begins at the symbolic address CONST.

External static areas

The EXTINIT operand of @DATA specifies the symbolic address of a data area in a separate data module. In the accessing procedure, the structure of this area must be described using a dummy section. The operand DSECT specifies the name of this dummy section.

Example

```
Name        | Operation | Operands
------------------------------------------------------------------
* Accessing procedure
EX2         | @ENTR     | TYP=I
            | .         |
            | .         |
            | @DATA     | CLASS=S,BASE=R5,EXTINIT=EXDATEN,DSECT=EX2DUMMY
            | .         |
            | .         |
            | @EXIT     |
            | @END      |
EX2DUMMY    | DSECT     |
E1          | DS        | CL5
            | .         |
            | .         |
EEND        | DS        | CL3

* Data module
EXDATEN     | CSECT     |
            | @ENTR     | TYP=D
D           | DS        | 0CL180
D1          | DC        | C'HALLO'
            | .         |
            | .         |
DEND        | DC        | C'END'
            | @END      |
```

Register 5 is allocated as the base address register for the static area of the EX2 procedure. The data area starts at the address EXDATEN. In the EX2 procedure, the dummy section EX2DUMMY describes the structure of the accessed data area.

9.4.2 Data areas of the automatic class

Data areas of the automatic class contain modifiable data. Storage space is requested by a procedure at runtime, and reserved in the procedure STACK. This procedure STACK is created on activation of each procedure and is automatically released on quitting the procedure with @EXIT (procedure-linked life).

If the procedure is called recursively, or executed by various users "simultaneously" (reentrant), several data areas exist side-by-side.

Data areas of the automatic class may only be requested in type M, E and I procedures.

There are two ways of requesting an automatic area:
– the data area is addressed via base register 13, along with the register save area of the procedure (LOCAL area request).
– the data area is addressed via a separate register, specified explicitly by the user (CLASS A area request).

LOCAL area request

When a procedure is opened, the procedure STACK, which contains the register save area (SAVAREA) and the LOCAL area, is automatically made available. The base address register for the procedure STACK and thus also for the SAVAREA and for the LOCAL area is register 13. The LOCAL area can be accessed after execution of the procedure heading.

The area is requested via the LOCAL operand of @ENTR (see section 10.2, formats 1 and 2 of @ENTR). The newly requested area cannot be initialized. Data in this memory area cannot be accessed from another procedure.

The structure of this memory area is described by a dummy section, which must be defined immediately after the static end of the procedure concerned. The dummy section must be defined with the @PAR macro (see section 10.2, format 3 of @PAR). The name of the dummy section must correspond to the specification in the LOCAL operand of @ENTR.

Example

```
Name        | Operation | Operands
------------+-----------+-------------------------
EX1         | @ENTR     | TYP=I,LOCAL=PRIVATE
            | .         |
            | .         |
            | @EXIT     |
            | @END      |
PRIVATE     | @PAR      | D=YES    (01)
P1          | DS        | CL5
            | .         |
            | .         |
PRIVATE     | @PAR      | LEND=YES              (02)
```

(01) Start of the dummy section
(02) End of the dummy section

@DATA CLASS=A request

For a data area requested with "@DATA CLASS=A", the register specified in the BASE operand of @DATA is allocated as the base address register.

Storage space is requested according to the length specified either directly in the LENGTH= operand or calculated using an assigned dummy section. If LENGTH= is specified, the symbolic addressing option ceases to be applicable.

The allocated dummy section describes the structure of the requested memory area. Its definition must always begin with a DSECT instruction. The dummy section must be terminated with the following assembler instruction:

```
Ldsect_name    EQU  *-dsect_name
```

Example

```
Name        | Operation | Operands
------------|-----------|-----------------------------
MAIN        | @ENTR     | TYP=M
            | @DATA     | CLASS=A,BASE=R7,DSECT=DUMMY
            | .         |
            | .         |
            | @EXIT     |
            | @END      |
DUMMY       | DSECT     |
D           | DS        | 0CL20
D1          | DS        | CL10
D2          | DS        | CL10
LDUMMY      | EQU       | *-DUMMY
```

Automatic areas requested with @DATA CLASS=A can be initialized. In other words, data already defined is copied into the memory area requested. This data may be in the same module as the procedure concerned, where it can be accessed with the operand INIT=. If the data to be copied is in another module, the name of the data area must be specified with EXTINIT=.

Example

```
Name        | Operation | Operands
------------|-----------|-----------------------------
MAIN        | @ENTR     | TYP=M
            | @DATA     | CLASS=A,BASE=R7,LENGTH=(R8),INIT=IDAT
            | .         |
            | .         |
            | @EXIT     |
IDAT        | DC        | C'ABC'
            | DC        | C'DEF'
            | .         |
            | .         |
            | @END      |
```

9.4.3 Data areas of the controlled class

Data areas of the controlled class contain modifiable data. Storage space is requested by a procedure at runtime and made available in the HEAP memory. It lies outside the module and can only be released explicitly by the user with @FREE (user-controlled life).

If the procedure is called recursively, or executed by various users "simultaneously" (reentrant), several data areas exist side-by-side for each execution of the @DATA macro.

Data areas of the controlled class can only be requested in type M, E and I procedures.

Data areas of the controlled class are requested with "@DATA CLASS=C", and addressed via the base address register which is explicitly specified in the BASE operand.

Storage space is requested according to the length specified either directly in the LENGTH= operand or calculated using an assigned dummy section. If LENGTH= is specified, the symbolic addressing option ceases to be applicable.

The allocated dummy section describes the structure of the requested memory area. Its definition must begin with a DSECT instruction. The dummy section must be terminated with the following assembler instruction:

```
Ldsect_name     EQU  *-dsect_name
```

Controlled areas requested with @DATA CLASS=C can be initialized. In other words, data already defined is copied into the requested memory area. This data may be in the same module as the procedure concerned, and is then accessed using the INIT= operand. If the data to be copied is in another module, the name of the data area must be specified with EXTINIT=.

The user must release the requested memory area with @FREE (see section 10.2). This can be done before the end of the procedure in order to save space and registers, or in a later procedure, if, e.g., the data is to be further processed.

Example

```
Name      | Operation | Operands
----------|-----------|-----------------------------------
MAIN      | @ENTR     | TYP=M
          | @DATA     | CLASS=C,BASE=R7,DSECT=DUMMY
          | .         |
          | .         |
          | @FREE     | BASE=R7
          | @EXIT     |
          | @END      |
DUMMY     | DSECT     |
D         | DS        | 0CL20
D1        | DS        | CL10
D2        | DS        | CL10
LDUMMY    | EQU       | *-DUMMY
```

9.4.4 Data areas of the based class

The based class is used to access a data area whose storage space is assigned in another dynamically higher-level or earlier procedure (@DATA CLASS=S, CLASS=A or CLASS=C).

A sub-procedure accesses an existing data area using "@DATA CLASS=B" (see 10.2, format 3 of @DATA). The BASE operand of this macro instruction specifies the base address register which has been allocated for this data area in a higher-level procedure. The register must be given the start address of the area in the higher-level procedure. This is done using @DATA CLASS=A, C or S or, for example, during parameter passing.

To enable symbolic addressing of the accessed data area, the DSECT operand must be specified in the @DATA macro. This either contains:

- the symbolic address of the accessed data area, or
- the name of a dummy section which is to describe the structure of the area. The dummy section must be closed with the following assembler instruction:

```
Ldsect_name     EQU  *-dsect_name
```

Example

```
Name        | Operation | Operands
------------+-----------+-------------------------------
EX1         | CSECT     |
DUMMY       | DSECT     |                                 ⎫
..          | DS        | ..                              ⎪
            | .         |                                 ⎬ (01)
            | .         |                                 ⎪
LDUMMY      | EQU       | *-DUMMY                         ⎭

* Higher-level procedure
EX1         | @ENTR     | TYP=M
            | @DATA     | CLASS=S,BASE=R7,INIT=GLOB     (02)
            | .         |
            | .         |
            | @PASS     | NAME=EX2                      (03)
            | .         |
            | .         |
            | @EXIT     |
            | @END      |
GLOB        | DS        | 0C
..          | .         | ..
            | .         |

* Sub-procedure
EX2         | @ENTR     | TYP=I
            | @DATA     | CLASS=B,BASE=R7,DSECT=DUMMY   (04)
            | .         |
            | .         |
```

(01) Definition of the dummy section

(02) Creation of a static area with register 7 as the base address register; the area is initialized with the data in GLOB.

(03) Calling the sub-procedure

(04) Access to the data area initialized in EX1 and which is addressed via R7. The structure of the memory area is described by the dummy section DUMMY for the EX2 procedure

9.5 Procedure linkage and parameter passing

Procedures are linked using @PASS (see section 10.2). This macro calls a subprocedure from a dynamically higher ranked procedure. The called procedure returns to the calling procedure with @EXIT.

When type M, E and I procedures are opened, the procedure STACK, which contains chaining information, the register (SAVAREA) and any LOCAL area, is made available automatically. The procedure STACK is addressed via register 13. The calling procedure transfers the address of the separate procedure STACK into register 13, the called procedure stores the registers in the procedure STACK of the calling procedure and loads them again before return.

Register 13 is not saved at register saving.

When type B, L and D procedures are called, no procedure STACK is made available. The user himself must provide register saving for these procedures. In particular, the procedure return register 14 must be saved for type B or L procedures.

Parameter passing

When calling another procedure, a procedure can pass parameters to the called procedure. A parameter is always one word long. It may contain an address or a value. Either the parameter itself or its address is transferred to the called procedure. A parameter list, containing the address or value for each value, is created for parameter passing.

Various forms of parameter passing are available. These take into account

- whether the parameter values to be passed during a call can be defined during programming (static passing) or whether they are only known at runtime (dynamic passing),
- whether, in dynamic passing, parameter addresses are combined into one list, whose address is transferred into register 1 (STANDARD interface) or whether up to four parameter addresses or values are transferred into registers 1 - 4 (OPTIMAL interface).

In static passing, the parameter list created for @PASS consists of constants generated at assembly time and retained during the entire life of the program. In dynamic passing, parameter lists are treated in the same way as modifiable data. Storage space is reserved in the procedure STACK of the calling procedure, and the parameter list is stored there when the procedure is called (@PASS).

Whether passing is to be in STANDARD or OPTIMAL form is specified

– in the calling procedure, in the PASS operand of the procedure call (@PASS),
– in the called procedure, in the PASS operand of the procedure heading (@ENTR).

Both specifications should correspond.

In the program "DEMOPARA" (see Appendix 11.5), the various options for parameter passing and acceptance are illustrated in an example.

The following table shows the possible combinations of the forms of parameter passing with those of parameter acceptance.

	Parameter acceptance:			
Parameter passing:	STANDARD interface (9.5.3.1)	OPTIMAL interface (9.5.3.2)	in formal parameter (9.5.3.3)	in formal par. in LOCAL (9.5.3.4)
STANDARD interface, static (9.5.1.1)	X	-	X	X
STANDARD interface, dynamic (9.5.1.2)	X	-	X	X
OPTIMAL interface, dynamic (9.5.2)	-	X	X	X

Table 9-3 Combinations of parameter passing and parameter acceptance

9.5.1 Parameter passing via the STANDARD interface

In parameter passing via the STANDARD interface, parameters are combined into a parameter list. The address of the parameter list is transferred into register 1.

9.5.1.1 Static parameter passing

In static parameter passing, the parameter list is generated with constant values at assembly time. It is retained until the end of the program. The parameter list is stored using the @PAR macro (format 1) in the calling procedure, i.e. it belongs to the corresponding module.

This form of parameter passing is allowed in type M, E, I, L and B procedures.

In the calling procedure

- The PAR operand of @PASS (format 2) specifies the name of the parameter list to be transferred. Register 1 is loaded with this address.
- The parameter list must be generated with @PAR (format 1) between @EXIT and @END. The name of @PAR specifies the name of the parameter list.
- The PLIST operand of @PAR (format 1) issues names to the address constants in the parameter list. If the PLIST operand is omitted, nameless constants are created.
- The VLIST operand of @PAR (format 1) contains the actual parameters. These are either the name of a field whose address is to be transferred, or a self-defining term.

  If there is no actual parameter, an extra comma must be entered in the VLIST operand.

In the called procedure

The following options are available for parameter acceptance:

- parameter reference via the parameter address list (STANDARD interface, see section 9.5.3.1)
- formal parameter acceptance, where fields in the called procedure correspond to formal parameters (see section 9.5.3.3) and
- formal parameter acceptance, where fields in the LOCAL area of the called procedure correspond to formal parameters (see section 9.5.3.4).

Example

```
Name        | Operation | Operands
------------+-----------+-------------------------------------------------
* Calling procedure
PRO1        | @ENTR     | TYP=M
            | @DATA     | CLASS=S,BASE=R6,INIT=PRO1DAT
            | .         |
            | .         |
            | @PASS     | NAME=PRO2,PAR=PARLIST
            | .         |
            | .         |
            | @EXIT     |
PARLIST     | @PAR      | PLIST=(PAR1,PAR2,PAR3),VLIST=(FIELD,27,SET)
            | @END      |
PRO1DAT     | DS        | 0H
FIELD       | DC        | C'ABC'
SET         | DC        | C'DEF'
            | .         |
            | .         |
```

Procedure PRO1 calls PRO2 and passes three parameters.
The address of FIELD, the value 27, and the address of SET are passed.

```
* Called procedure
PRO2        | @ENTR     | TYP=I,LOCAL=IN,PLIST=(IN1,IN2,IN3)
            | @DATA     | CLASS=S,BASE=R7,INIT=PRO2DAT
            | .         |
            | .         |
            | L         | R8,IN3
            | MVC       | FIELDB,0(R8)
            | .         |
            | .         |
            | @EXIT     |
            | @END      |
IN          | @PAR      | D=YES,LEND=YES,PLIST=(IN1,IN2,IN3)
            | .         |
            | .         |
PRO2DAT     | DS        | 0H
FIELDA      | DS        | CL3
FIELDB      | DS        | CL3
            | .         |
            | .         |
```

PRO2 accepts three parameters into its LOCAL area and transfers the contents of SET to FIELDB.

9.5.1.2 Dynamic parameter passing

In dynamic parameter passing, the parameter list is created at runtime and stored in the save area of the calling procedure. On return from this procedure, the memory area is released. The parameter list is formatted via the PLIST operand of @PASS (format 2) in the calling procedure.

Dynamic parameter passing is only possible in type M, E and I procedures.

In the calling procedure

- The MAXPRM operand of @ENTR (format 1 and 2) specifies the maximum number of parameters the procedure may pass. Storage space is reserved accordingly in the LOCAL area of the calling procedure for the maximum size of the parameter list.
- The PLIST operand of @PASS (format 3) specifies the names of the fields or registers whose address is to be transferred to the calling procedure (actual parameters).
- The PASS operand of @PASS (format 3) specifies that parameter passing is to be done via the STANDARD interface.

In the called procedure

The following options are available for acceptance of parameters:

- parameter reference via the parameter list address (STANDARD interface, see section 9.5.3.1),
- formal parameter acceptance, where fields in the called procedure correspond to formal parameters (see section 9.5.3.3), and
- formal parameter acceptance, where fields in the LOCAL area of the called procedure correspond to formal parameters (see section 9.5.3.4).

Example

```
Name        | Operation | Operands
------------+-----------+-----------------------------------------
* Calling procedure
PRO1        | @ENTR     | TYP=M,MAXPRM=2
            | @DATA     | CLASS=A,BASE=R6,DSECT=DUMMY,INIT=PRO1DAT
            | .         |
            | .         |
            | @PASS     | NAME=PRO2,PLIST=(D1,D2),PASS=STA
            | .         |
            | .         |
            | @EXIT     |
PRO1DAT     | DS        | 0H
FIELD       | DC        | C'ABC'
SET         | DC        | C'DEF'
            | @END      |
DUMMY       | DSECT     |
D1          | DS        | CL3
D2          | DS        | CL3
            | .         |
            | .         |
PRO1        | CSECT     |
```

Procedure PRO1 calls PRO2 and passes two parameters. The fields whose addresses are to be transferred are defined in DUMMY, and initialized with the values from PRO1DAT.

```
* Called procedure
PRO2        | @ENTR     | TYP=I,LOCAL=IN,PLIST=(IN1,IN2),PASS=STA
            | @DATA     | CLASS=S,BASE=R7,INIT=PRO2DAT
            | .         |
            | .         |
            | L         | R8,IN2
            | MVC       | FIELDB,0(R8)
            | .         |
            | .         |
            | @EXIT     |
            | @END      |
IN          | @PAR      | D=YES,LEND=YES,PLIST=(IN1,IN2)
            | .         |
            | .         |
PRO2DAT     | DS        | 0H
FIELDA      | DS        | CL3
FIELDB      | DS        | CL3
            | .         |
            | .         |
```

PRO2 accepts two parameters into its LOCAL area and passes the contents of D2 to FIELDB.

9.5.2 Parameter passing via the OPTIMAL interface

Parameters can only be passed dynamically via the OPTIMAL interface. Here, the parameters are passed directly to registers 1 to 4. The addresses of the fields or values which are to be passed to the called procedure are stored in these registers.

If there are more than 4 parameters, the first three are passed to registers 2 to 4. A parameter list is created for the remaining parameters, and its address is loaded into register 1. In other words, three parameters are passed via the OPTIMAL interface, and the others via the STANDARD interface.

Dynamic parameter passing is only possible in type M, E and I procedures.

For linkage of programs in other programming languages, this form of parameter passing is **not** permitted.

In the calling procedure

- The PLIST operand of @PASS (format 3) specifies the actual parameters, i.e. the addresses of the fields or or registers which are to be passed to the called procedure.
- The PASS operand of @PASS (format 3) specifies that passing is to be done via the OPTIMAL interface.

In the called procedure

The following options are available for parameter acceptance:

- direct access to parameters via registers 1 to 4 (OPTIMAL interface, see 9.5.3.2),
- formal parameter acceptance, where fields in the called procedure correspond to formal parameters (see section 9.5.3.3) and
- formal parameter acceptance, where fields in the LOCAL area of the called procedure correspond to formal parameters (see section 9.5.3.4).

Example

```
Name        | Operation | Operands
------------+-----------+-------------------------------------------------
* Calling procedure
PRO1        | @ENTR     | TYP=M
            | @DATA     | CLASS=A,BASE=R6,DSECT=DUMMY,INIT=PRO1DAT
            | .         |
            | .         |
            | @PASS     | NAME=PRO2,PLIST=(D1,D2),PASS=OPT
            | .         |
            | .         |
            | @EXIT     |
PRO1DAT     | DS        | 0H
FIELD       | DC        | C'ABC'
SET         | DC        | C'DEF'
            | @END      |
DUMMY       | DSECT     |
D1          | DS        | CL3
D2          | DS        | CL3
            | .         |
            | .         |
PRO1        | CSECT     |
```

Procedure PRO1 calls PRO2 and passes two parameters. The fields whose addresses are to be passed are defined in DUMMY, and initialized with the values from PRO1DAT.

```
* Called procedure
PRO2        | @ENTR     | TYP=I,LOCAL=IN,PLIST=(IN1,IN2),PASS=OPT
            | @DATA     | CLASS=S,BASE=R7,INIT=PRO2DAT
            | .         |
            | .         |
            | L         | R8,IN2
            | MVC       | FIELDB,0(R8)
            | .         |
            | .         |
            | @EXIT     |
            | @END      |
IN          | @PAR      | D=YES,LEND=YES,PLIST=(IN1,IN2)
            | .         |
            | .         |
PRO2DAT     | DS        | 0H
FIELDA      | DS        | CL3
FIELDB      | DS        | CL3
            | .         |
            | .         |
```

PRO2 accepts two parameters in its LOCAL area and moves the contents of D2 to FIELDB.

9.5.3 Parameter acceptance

Depending on the form of transfer and procedure type, there are various options for the acceptance of parameters.

9.5.3.1 Parameter acceptance via the STANDARD interface

In STANDARD acceptance, register 1 always contains the parameter list address. The called procedure can utilize this STANDARD interface and reference parameters via the parameter address.

This format is allowed for all procedure types.

Example

```
Name          | Operation | Operands
--------------------------------------------------------------------
* Calling procedure
PRO1          | @ENTR     | TYP=M
              | @DATA     | CLASS=S,BASE=R6,INIT=PRO1DAT
              | .         |
              | .         |
              | @PASS     | NAME=PRO2,PAR=PARLIST
              | .         |
              | .         |
              | @EXIT     |
PARLIST       | @PAR      | PLIST=(PAR1,PAR2,PAR3),VLIST=(FIELD,27,SET)
              | @END      |
PRO1DAT       | DS        | 0H
FIELD         | DC        | C'ABC'
SET           | DC        | C'DEF'
              | .         |
              | .         |
```

Procedure PRO1 calls PRO2 and passes three parameters (static passing).
The address of FIELD, the value 27, and the address of SET are passed.

```
Name          | Operation | Operands
--------------+-----------+-----------------------------------
* Called procedure
PRO2          | @ENTR     | TYP=I
              | @DATA     | CLASS=S,BASE=R7,INIT=PRO2DAT
              | .         |
              | .         |
              | L         | R8,0(0,R1)
              | MVC       | FIELDA,0(R8)
              | L         | R8,8(0,R1)
              | MVC       | FIELDB,0(R8)
              | .         |
              | .         |
              | @EXIT     |
              | @END      |
              | .         |
              | .         |
PRO2DAT       | DS        | 0H
FIELDA        | DS        | CL3
FIELDB        | DS        | CL3
              | .         |
              | .         |
```

PRO2 accepts the parameters and moves the contents of FIELD to FIELDA, and the contents of SET to FIELDB.

9.5.3.2 Parameter acceptance via the OPTIMAL interface

In OPTIMAL acceptance, the parameters are passed into registers 1 to 4. The called procedure can immediately utilize this OPTIMAL interface and access the parameters via register 1 to 4. If there are more than four parameters, the called procedure must also take care of the corresponding management.

In the called procedure, the @ENTR macro (format 2) of the PASS operand must specify that passing is effected via the OPTIMAL interface.

Example

```
Name        | Operation | Operands
------------------------------------------------------------------
* Calling procedure
PRO1        | @ENTR     | TYP=M
            | @DATA     | CLASS=A,BASE=R6,DSECT=DUMMY,INIT=PRO1DAT
            | .         |
            | .         |
            | @PASS     | NAME=PRO2,PLIST=(D1,D2),PASS=OPT
            | .         |
            | .         |
            | @EXIT     |
PRO1DAT     | DS        | 0H
FIELD       | DC        | C'ABC'
SET         | DC        | C'DEF'
            | @END      |
DUMMY       | DSECT     |
D1          | DS        | CL3
D2          | DS        | CL3
            | .         |
            | .         |
PRO1        | CSECT     |
```

Procedure PRO1 calls PRO2 and passes two parameters. The fields whose addresses are to be transferred are defined in DUMMY, and are initialized with the values from PRO1DAT.

```
Name        | Operation | Operands
------------------------------------------------------------
* Called procedure
PRO2        | @ENTR     | TYP=I,PASS=OPT
            | @DATA     | CLASS=S,BASE=R7,INIT=PRO2DAT
            | .         |
            | .         |
            | MVC       | FIELDB,0(R2)
            | .         |
            | .         |
            | @EXIT     |
            | @END      |
            | .         |
            | .         |
PRO2DAT     | DS        | 0H
FIELDA      | DS        | CL3
FIELDB      | DS        | CL3
            | .         |
            | .         |
```

PRO2 accepts two parameters and moves the contents of D2 to FIELDB.

9.5.3.3 Formal parameter acceptance

In this case, those fields defined in the called procedure correspond to the formal parameters.

This form of parameter acceptance is not practical for type B or L procedures. They should also not be used for READ-ONLY procedures and in recursive calls.

In the called procedure

- The PLIST operand of @ENTR (format 2) must contain the list of formal parameters. The parameter list entries are accepted into these fields.
- The PASS operand of @ENTR (format 2) must specify whether parameters are accepted via the OPTIMAL or the STANDARD interface.
- The data fields specified in the PLIST operand must be defined in the procedure by the user. This is only possible with the aid of @DATA CLASS=S.

Example

```
Name        | Operation | Operands
------------+-----------+-----------------------------------------
* Calling procedure
PRO1        | @ENTR     | TYP=M
            | @DATA     | CLASS=S,BASE=R6,INIT=PRO1DAT
            | .         |
            | .         |
            | @PASS     | NAME=PRO2,PAR=PARLIST
            | .         |
            | .         |
            | @EXIT     |
PARLIST     | @PAR      | PLIST=(PAR1,PAR2,PAR3),VLIST=(FIELD,27,SET)
            | @END      |
PRO1DAT     | DS        | 0H
FIELD       | DC        | C'ABC'
SET         | DC        | C'DEF'
            | .         |
            | .         |
```

The procedure PRO1 calls PRO2 and passes three parameters (static passing). The address of FIELD, the value 27 and the address of SET are passed.

```
Name        | Operation | Operands
------------+-----------+-----------------------------------
* Called procedure
PRO2        | @ENTR     | TYP=I,PLIST=(IN1,IN2,IN3)
            | @DATA     | CLASS=S,BASE=R7,INIT=PRO2DAT
            | .         |
            | .         |
            | L         | R8,IN3
            | MVC       | FIELDB,0(R8)
            | .         |
            | .         |
            | @EXIT     |
            | @END      |
            | .         |
            | .         |
PRO2DAT     | DS        | 0H
FIELDA      | DS        | CL3
FIELDB      | DS        | CL3
IN1         | DS        | F
IN2         | DS        | F
IN3         | DS        | F
```

PRO2 accepts three parameters and moves the contents of SET to FIELDB.

9.5.3.4 Formal parameter acceptance in the LOCAL area.

Here, fields which are in the LOCAL area of procedure STACKS (see 9.4.2) of the called procedure correspond to formal parameters. The structure of this area describes a dummy section which must match the structure of the parameter list. In the called procedure, a formal parameter defined in the dummy section, or the corresponding register, therefore designates the address of the constants from the calling procedure.

This form of parameter acceptance is allowed only for type E or I procedures. It is mandatory for READ-ONLY procedures, and in recursive procedure calls.

In the called procedure

- A LOCAL area in the procedure STACK in which parameters are to be accepted must be requested via the LOCAL operand of @ENTR (format 2).
- The PLIST operands of @ENTR (format 2) must contain the list of formal parameters. Entries from the parameter list must be accepted in these fields or registers.
- The PASS operand of @ENTR (format 2) must specify whether parameters are to be passed via the OPTIMAL or the STANDARD interface.
- A dummy section must be generated immediately after the procedure using @PAR (format 2). The name of this dummy section must match the name in the LOCAL operand of @ENTR.
- The PLIST operand of @PAR specifies the names of the formal parameter in the dummy section, which describe the structure of the local area. The PLIST operand of @PAR must match the PLIST operands of @ENTR.

Example see section 9.5.1, "Static parameter passing", or 9.5.2, "Dynamic parameter passing".

9.6 ILCS interface for structured programming

With Version 1.1A of ASSEMBH the macros for structured programming (@-macros, see chapter 10) and the assembler runtime system (ASSEMBH-RTS) have been extended so as to allow the user additionally to write programs with ILCS capability in assembler.
The following facilities are available in the ILCS environment:

The following ILCS functions are supported by means of new operands in the @ENTR macro for the main procedure:

- enable user-own routines for reserving and releasing memory space for stack and heap
- enable user-own termination routines
- specify minimum stack extent size

In addition to these adaptations to the ILCS standard, new structure macros offer the user access to the following procedure-independent functions of ILCS:

- event handling
- contingency handling
- interrupt handling (STXITs)

The new macros are intended to be used in ILCS procedures instead of the necessary system macro calls for the above types of handling.

The following functions are supported by additional new macros:

- program mask handling
- setting of monitoring job variables
- language initialization in the case of dynamically loaded modules

The new @ macros generate the call for corresponding entries in the standard event handler (SEH), the standard contingency handler (SCH), and the standard STXIT handler (SSH) of the ILCS interface.

If the user does not wish to write an ILCS procedure, the macros can be called in the same way as before. This means that existing sources that are not to be converted to ILCS do not have to be modified or re-assembled.

9.6.1 Procedure linkage

The extensions to the @ENTR, @PASS, @PAR and @EXIT macros enable the procedure interface to be converted to ILCS conventions.

This affects the procedure prolog, the procedure epilog with returned function value and return code, and the procedure call with parameter passing.

Register conventions (ILCS interface and non-ILCS interface)

The conventions of the ILCS and non-ILCS interfaces are compared in the following table in respect of

- register usage at procedure call with/without parameters
- register restoring at procedure end with/without function value and return code
- long jump over several procedure levels at procedure end.

Register usage at procedure call

ILCS	Non-ILCS
R15 contains the address of the called procedure	as for ILCS
R14 contains the return address	as for ILCS
R1 contains the address of the parameter list for STANDARD parameter passing; the left bit is set in the final word of the list	as for ILCS
R0 contains the number of actual parameters passed	Number of parameters is not contained in R0; the left bit is set in the final word of the list
OPTIMAL parameter passing is not permitted	OPTIMAL parameter passing permitted in addition to STANDARD
"Call by reference" parameter passing, i.e. R1 contains the address of a parameter address list	Both "call by value" and "call by reference" parameter passing are possible.

Register restoring at procedure end

ILCS	Non-ILCS
R2 through R14 are restored	No specification in @EXIT/@ENTR: R2 through R14 restored R0, R1: function value R15: return code
RETURNS=YES (@ENTR): R0 and R1: function value	RETURNS=YES (@ENTR): R2 through R14 and R0 restored R1: function value R15: return code
RETURNS=NO (@ENTR): R0 and R1: undefined	RETURNS=NO (@ENTR): R0 through R14 restored R15: return code RESTORE=MIN (@EXIT): R7 through R14 restored R0 through R6: function value R15: return code PROG=FORTRAN (@EXIT): R2 through R14 restored R1: not evaluated by FORTRAN R0: return code

Long jump

ILCS	Non-ILCS
Not allowed	Via @EXIT TO = possible, executable only in non-ILCS environment

Parameter passing

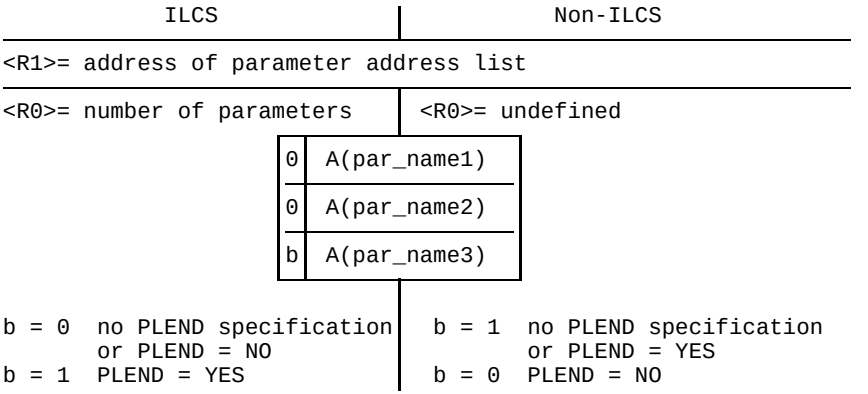
There are two options for the passing and acceptance of parameters:
- in STANDARD form
- through "call by reference"
  The user must specify in the list of actual parameters (@PAR/@PASS macro) the addresses of the parameter values to be passed, and not the values themselves.

Note:
In both static and dynamic parameter passing, the left bit in the last parameter in the parameter list will be set in accordance with the PLEND parameter.

The following are illegal:
- in @ENTR: the parameter PASS=OPTIMAL
- in @PASS: the parameter PASS=OPTIMAL, and in static parameter passing the parameter PAR=(<register>).
- in @EXIT: the parameter RESTORE=MIN, TO=<proc.name> and PROG=FORTRAN.

Examples of parameter passing

1. "Call by reference"
   @PASS call destination, PLIST=(par_name1, par_name2, par_reg), PASS=STANDARD in which <par_reg>=par_name3



2. "Call by reference" and "Call by value"
   @PASS call destination, PLIST=(par_name1,2, par_name3), PASS=STANDARD

ILCS	Non-ILCS
Direct value allowed only if the calling procedure interprets this as an absolute address	Direct value always allowed; it is interpreted as an absolute address or as a direct value
<R1>= address of parameter address list	
<R0>= number of parameters	<R0>= undefined

0	A(par_name1)
0	A(2)
b	A(par_name3)

b = 0 no PLEND specification or PLEND = NO b = 1 PLEND = YES	b = 1 no PLEND specification or PLEND = YES b = 0 PLEND = NO

3. "Call by reference" and missing parameters
   @PASS call destination, PLIST=(par_name1,, par_name3), PASS=STANDARD

ILCS	Non-ILCS
Not permittted MNOTE (Significant Error)	Not permitted Mnote and flag (Severe)
<R1>= undefined	<R1>= address of parameter address list
<R0>= undefined	

0	A(par_name1)
0	A(0)
1	A(par_name3)

9.6.2 Activating user-own routines

With ILCS the user can activate user-own routines during initialization:
- for reserving and releasing memory space for stack and heap
- for termination handling
- to specify the minimum stack extent size

These routines must comply with ILCS conventions. They are valid across the entire ILCS environment and can be defined only in the main procedure @ENTR with TYP = M (see @ENTR, operand STREQ ff; chapter 10).

9.6.3 Event handling

With the standard event handler (SEH) ILCS provides a routine for coordinating events occurring during ILCS programs.

The following events are handled by ILCS:
- STXIT events in the BS2000 STXIT classes PROCHK and ERROR.
- Non-STXIT events that are not covered by BS2000 (e.g. OPEN errors when accessing files).

STXIT events are supplied by the system, whereas non-STXIT events must be signaled to the standard event handler by a procedure.

The event handling routines for the various event classes are enabled by means of the @ENTR macro with appropriate parameters.

The event handling routines specified by the user in the procedure prolog are enabled dynamically when a procedure is called, and disabled when the procedure is exited.

These routines cannot be explicitly enabled or disabled by the user.

Control always passes to the standard event handler when an event occurs for which the user has not registered any STXIT routine with the standard STXIT handler, or if the standard event handler has not been terminated.

A separate stack and heap is reserved and created by ILCS for each STXIT or contingency process initiated by an event. In the case of STXIT events in BS2000, the status of the interrupt point is recorded by the standard event handler. When a non-STXIT event is signaled, the user passes to the standard event handler a parameter block containing a description of the interrupt point.

The standard event handler searches in a separate appended event handler list for the address of the handling routine responsible for this event, starting the search from the current save area within the save area chain. If the routine is found, it is called by the standard event handler in accordance with ILCS conventions; the context of the interrupt point, extended by an additional parameter block, is then passed as a parameter to the routine by the standard event handler.

The following individual functions of the standard event handler can be called using new structure macros:

- enable event handler routines (see @ENTR, operands ABKR, PROCHK, ERROR and OTHEVT; chapter 10)
- signal non-STXIT event (see @EVTOE, chapter 10)

9.6.4 Contingency handling

As well as event handling, ILCS provides a standard contingency handler (SCH), a routine that enables users to use contingency processes in ILCS programs.

These contingency processes are user-defined external procedures that are managed by the standard contingency handler and called in accordance with ILCS conventions.

A contingency routine can be enabled and disabled using new structure macros.

When an event occurs it still has to be signaled by the user using the system macro POSSIG (post signal) for an event identifier. The request for a signal must be made using the system macro SOLSIG (solicit signal).

When an event occurs, such as a signal to an event ID, it first activates a standard contingency handler routine assigned to the user routine; this establishes the environment necessary for the handling routine, including a separate heap and a stack for the save areas of the user task.

The standard contingency handler then calls the user contingency routine in accordance with ILCS conventions.

The following individual functions can be accessed via the new macro interface:

- enable contingency routine (see @CONEN, chapter 10)
- disable contingency routine (see @CONDI, chapter 10)

9.6.5 STXIT handling

Users may work with their own STXIT routines within ILCS programs on the procedure level. For this purpose ILCS provides the standard STXIT handler (SSH), a routine in which user-enabled handling routines are called in accordance with ILCS conventions when STXIT events occur.

These routines are enabled and disabled in the ILCS environment by new structure macros with appropriate parameters.

All STXIT events defined in BS2000 are supported by ILCS.

In structured ILCS programs, explicit calls of BS2000 macros for STXIT handling can be replaced by the new structure macros. In the ILCS environment these ensure that the STXIT handler is called in accordance with ILCS conventions.

Control always passes to the standard STXIT handler when user-own STXIT routines are enabled for an STXIT event; a separate stack and heap are set up for an STXIT process.

The enabled STXIT routines are called by the standard STXIT handler in the defined BS2000 processing sequence in accordance with ILCS conventions.

When all the STXIT routines have been executed, the standard STXIT handler releases the separate stack and heap and reestablishes the environment of the interrupted procedure.

The following ILCS functions are accessible using this macro:

- enable STXIT routine (see @STXEN, chapter 10)
- disable STXIT routine (see @STXDI, chapter 10)

9.6.6 Setting the program mask

With the new structure macro @SETPM (see chapter 10) the user can dynamically modify the program mask, including the ID code, in procedures so that events occurring during certain mathematical operations do not cause the system to interrupt the program.

These events can include:

- fixed-point overflow
- decimal overflow
- exponent underflow
- mantissa=0

In structured ILCS programs this macro replaces the BS2000 command for setting the program mask.

9.6.7 Setting the MONJV value in the PCD

Using the new structure macro @SETJV (see chapter 10) the user can define a value within procedures that is entered in the corresponding field in the PCD (see "ASSEMBH User Guide" [1], "ILCS data structures"). With this MONJV value as the parameter, the system macro TERM is called by ILCS.

9.6.8 Language initialization for dynamically loaded modules

Modules can be dynamically loaded within structured ILCS programs. The @ININ macro (see chapter 10) must then be called by ILCS:
ILCS checks whether language initialization is necessary. If it is, ILCS activates the initialization routine unless it has already been called.

10 Predefined macros for structured programming

General programming notes

Macro instruction format

Predefined macros for structured programming are referenced via their corresponding macro call. These macro instructions have an "@" as their first character, which is why they are also known as "@ macros".

The format of predefined macro instructions is the same as that of the macro instructions described in 7.1. The rules listed there also apply in this case.

The name, operation and operand entries of predefined macro instructions can be generated under certain conditions using **variable symbols** (see chapter 6, "Variable symbols").

As a rule, a sequence symbol in the name entry can be specified in a macro instruction. Within the context of structured programming, this must be avoided at all costs.

Macro instruction operands

Positional operands are assigned values according to their position in the operand entry. They must therefore be in the same sequence as that specified in the format.

Keyword operands are identified by the equals sign (=). They are assigned a value in the macro instruction via the keyword. Keyword operands may appear in any order in the macro instruction (see 7.1.1, "Keyword and positional operands").

Names

In structured programming there are predefined names for registers. General-purpose registers can be referenced with the names R0, R1, ..., R15, and floating-point registers, with FA, FB, FC and FD, without any need for explicit assignment beforehand.

In addition to these predefined names, all names beginning with "@" or "R@" are reserved for ASSEMBH and must not be used anywhere else.

@AND Logical AND

Function

@AND implements the logical AND operation in compound conditions.

Format

Name	Operation	Operands
[name]	@AND	cond_sym

name Name

cond_sym Predefined or user-own condition symbol (see section 9.2.2).

Programming notes

Basically, the call to the @AND macro must be followed by a condition code setting machine instruction (see "Assembler Instructions" manual [3]).

Example

```
Name        Operation   Operands
            .
            .
            @IF         ZE
            C           R4,ZERO          (01)
            @AND        NZ
            LTR         R5,R5            (02)
            @THEN
            ST          R1,POINT
            @BEND
            .
```

The @THEN branch is only executed if the *first condition* (01) and the *second condition* (02) hold true.

>>>>> See also @OR and @TOR

@BEGI Sequence

Function

@BEGI forms the entry point of a sequence structure block.

Format

Name	Operation	Operands
[name]	@BEGI[N]	

@BEND Structure block end

Function

@BEND defines the block end for all types of structure block.

Format

Name	Operation	Operands
[name]	@BEND	

>>>>> See also @BEGI, @CASE, @CAS2, @CYCL, @IF, @THRU, @WHIL

@BREA Termination of a loop

Function

@BREA defines an exit from a loop with unrestricted terminal condition or a count loop with unrestricted terminal condition.

Format

Name	Operation	Operands
[name]	@BREA[K]	

Description

@BREA appears in conjunction with the macro instruction @WHEN. If the condition defined via @WHEN holds true, the structure block is quit with @BREA. If the condition does not hold true, the instruction statement following @BREA is executed.

Example

```
Name        Operation     Operands
LOOP        @CYCL
            .
            .
            @WHEN         NE
            CLI           OK,C'Y'
            @BREA                          (01)
            .
            .
            @WHEN         LZ
            SR            R7,R5
            @BREA                          (02)
            .
            .
            @BEND
```

The example shows a loop with two terminal conditions and the two possible exits (01) and (02). The structure block is quit the first time one of the two conditions holds true.

>>>>> See also @CYCL and @WHEN

@CASE Case differentiation by number

Function

@CASE forms the heading of a multiple branch. The sub-block to be executed is selected by specifying its number.

Format

Name	Operation	Operands
[name]	@CASE	(reg)

name Name
reg General-purpose register containing one of the following terms as a positive absolute expression:
- decimal self-defining term or
- predefined name of a register or
- name to which a corresponding self-defining term was assigned.

Description

The contents of reg specify which sub-block in a case differentiation is to be executed. There may be a maximum of 90 sub-blocks. The sub-block specified is selected directly.

Register 0 may not be used.

Programming notes

1. reg must be loaded with the number of the relevant sub-block prior to calling @CASE.
2. When using a general-purpose register for reg, the rules regarding registers must be adhered to (see "Use of registers" in section 11.3 of the appendix).
3. If the content of the @CASE register is less than 1 or greater than the number of sub-blocks defined,
   - the last sub-block is executed if CHECK=ON is set in the procedure heading (see 10.2, @ENTR);
   - program errors occur if CHECK=OFF is set in the procedure heading.

Example

```
Name      | Operation | Operands
----------+-----------+-----------------------------
          | L         | R6,T2
          | @CASE     | (R6)
          | @BEGI     |                  ┐
          | .         |                  } 1st sub-block
          | .         |                  ┘
          | @BEND     |
          | @BEGI     |                  ┐
          | .         |                  } 2nd sub-block
          | .         |                  ┘
          | @BEND     |
          | .         |                  ┐ other sub-blocks
          | .         |                  } possible
          | @BEND     |
          | .         |
          | .         |
T1        | DC        | F'1'
T2        | DC        | F'2'
          | .         |
          | .         |
```

In the example, register 6 is used for the @CASE branch. The content of register 6 is 2 in this case, hence the second sub-block is executed.

>>>>> See also @BEGI and @BEND

@CAS2 Case differentiation by comparison

Function

@CAS2 forms the heading of a multiple branch. The sub-block to be executed is selected by specifying a selector.

Format

Name	Operation	Operands
[name1]	@CAS2	{name2 / literal / (reg)} [,COMP=instr]

name1 Name
name2 Name of the field containing the selector
literal Literal specifying the selector directly (see 2.5.3).
reg General-purpose register containing the selector, either
- decimal self-defining term or
- predefined name of a register or
- name to which an appropriate self-defining term has been allocated

instr Machine instruction which sets a condition code (see "Assembler Instructions" manual [3]).

Description

The selector in the @CAS2 macro is compared with the comparands of the @OF macro. If selector and comparand match, the appropriate sub-block is executed. The number of sub-blocks is unrestricted.

A non-standard type of comparison can be selected with the COMP=instr.

Example

```
Name        | Operation | Operands
------------+-----------+-----------------------------
            | @CAS2     | TEST
            | @OF       | =X'0005'
            | .         |                          } 1st
            | .         |                          } sub-block
            | @OF       | =X'0006'
            | .         |                          } 2nd
            | .         |                          } sub-block
            | @OFRE     |
            | .         |                          } remainder
            | .         |                          } sub-block
            | @BEND     |
```

The field TEST contains the selector. If the selector and the first comparand (X'0005') match, the first sub-block is executed. If the selector and the second comparand (X'0006') match, the second sub-block is executed. In all other cases, the remainder sub-block is processed.

>>>>> See also @OF, @OFRE and @BEND

@CONDI Disable contingency routine

This macro instruction is only allowed in procedures with ILCS=YES.

Function

@CONDI (**Con**tingency **di**sable) deactivates a contingency routine
(see also section 9.6.4, "Contingency handling").

Format

Name	Operation	Operands
[name]	@CONDI	CONID={symb addr / (reg)}

name Name
symb addr Symbolic address of a field
reg General-purpose register containing the address of the ID code.

Description

CONID= Identifier of the routine

symb addr
Symbolic address of a field of length 4 aligned on a fullword boundary and containing the ID code. The ID code is supplied by the @CONEN macro.

(reg) Register containing the address of the ID code.

The inline code of the macro copies the specifications from the instruction into an ILCS-conforming parameter block; these specifications are used to activate the corresponding entry in the standard contingency handler for disabling the contingency routine.

When the standard contingency handler returns control to the calling procedure, R15 contains a return code which indicates whether or not the function has been executed and which errors, if any, occurred.

@CONEN Enable contingency routine

This macro instruction is allowed only in procedures with ILCS=YES.

Function

@CONEN (**Con**tingency **en**able) activates a contingency routine (see also section 9.6.4, "Contingency handling").

Format

Name	Operation	Operands
[name]	@CONEN	CONAME={name1 | symb addr1 | (reg1)} [CONLEN=length] CONID={symb addr2 | (reg2)} CONADR={symb addr3 | (reg3)} CONMSG={symb addr4 | (reg4)} [CONLEV={level | (reg5)}]

name Name
name1 Name of the contingency routine

symb addr1...symb addr4
Symbolic addresses of fields

reg1...reg4
General-purpose registers containing the address of a field.

length Length in bytes
level Decimal self-defining term
reg5 General-purpose register containing a priority specification.

Description

CONAME= name of contingency routine

name1 The name of the contingency routine consists of a string of up to 54 characters. Names are terminated with a blank.

symb addr1
The name of the contingency routine is contained in a field whose symbolic address is specified.

(reg1) The address of the name for the contingency routine is contained in a register.

CONLEN= length of routine name

length Length of the name in bytes; required only when passing an address for the name of the contingency routine.
Default if CONAME=symb addr1: length attribute of the symbolic address
Default if CONAME=(reg1): 54 bytes

CONID= returned ID code

symb addr2
Specifies the symbolic address of a field of length 4 aligned on a fullword boundary, in which the ID code is to be returned.

(reg2) Specifies a register containing the address of a field of length 4 aligned on a fullword boundary.

CONADR= start address of contingency routine

symb addr3
The start address of the contingency routine is contained in a field aligned on a fullword boundary, whose symbolic address is specified.

(reg3) The start address of the contingency routine is contained in a register.

CONMSG= user message

symb addr4
The user message to the contingency routine is contained in a field of length 4 aligned on a fullword boundary with the given symbolic address.

(reg4) The address of the user message to the contingency routine is contained in a register.

CONLEV= contingency routine priority

level The priority of the contingency routine is specified as a decimal self-defining term in the range 1 - 126.
The default is 1.

(reg5) The priority specification of the contingency routine is contained in a register.

The inline code of the macro copies the specifications from the macro call into an ILCS-conforming parameter block; these specifications are used to activate the corresponding entry in the standard contingency handler for enabling the contingency routine.

When the standard contingency handler returns control to the calling procedure, R15 contains a return code which indicates whether or not the function has been executed and which errors, if any, occurred.

@CYCL Loop heading

Function

@CYCL forms the heading of a loop construction. The number of executions is determined by a duplication factor or by a terminal condition.

Format 1: Loop with unrestricted terminal condition

Name	Operation	Operands
[name]	@CYCL[E]	

>>>>> See also @BREA, @WHEN and @BEND

Format 2: Count loop and count loop with unrestricted terminal condition

Name	Operation	Operands
[name]	@CYCL[E]	(reg)

name Name

reg General-purpose register; positive absolute expression, either
- decimal self-defining term or
- predefined name of a register or
- name to which an appropriate self-defining term has been assigned

Description

The contents of reg specify the number of repetitions of the loop sub-block. The maximum value of the repeat count is X'FFFFFFFF'.

reg is decremented by 1 after each execution, and checked for 0. If 0 is reached, the structure block is quit.

Programming notes

1. When specifying reg, the rules for registers must be adhered to (see "Use of registers" in section 11.3 of the appendix).
2. reg may not be altered in the loop sub-block.

Example

The example shows a count loop with unrestricted terminal condition. In this case, the loop is executed ten times, provided the termination condition does not occur beforehand.

```
Name      | Operation | Operands
----------+-----------+-----------------
          | L         | R7,NR              (01)
LOOP      | @CYCL     | (R7)               (02)
          | .         |
          | .         |
          | @WHEN     | EQ                 }
          | CLC       | TEST,END           } (03)
          | @BREA     |                    }
          | .         |
          | .         |
          | @BEND     |
          | .         |
          | .         |
NR        | DC        | F'10'
TEST      | .         | .
END       | .         | .
```

(01) Register 7 is loaded with the duplication factor
(02) Start of loop
(03) Termination condition and loop exit

>>>>> See also @BREA, @WHEN and @BEND

@DATA Data access and memory request

Function

@DATA implements access to user data and the request for the storage space required for it. Depending on the storage class specified, appropriate instruction statements are generated by the @DATA macro.

Format 1: Class A and C data areas

Name	Operation	Operands
[name]	@DATA	CLASS={A|C},BASE=reg1 [,DSECT=dsect_name] {[,LENGTH={val|(reg2)}]} [{[,INIT={int_name|(reg3)}]|,EXTINIT=ext_name}]

name Name

reg1, reg2, reg3
General-purpose registers; positive absolute expressions, either
- decimal self-defining terms or
- predefined register names
- names to which an appropriate self-defining term has been allocated

dsect_name
Name of a dummy section, 7 characters maximum

val Decimal self-defining term

int_name, ext_name
Names of data areas

Description

CLASS=A

This is a data area of class automatic.
CLASS=A is not allowed in type B, L or D procedures, or in any procedures in which "ENV=C" is set in @ENTR.

CLASS=C

This is a controlled class data area.
CLASS=C is not allowed in type B, L or D procedures.

BASE=reg1

specifies the base address register to be used.

When calling @DATA, the register named is loaded with the address of the memory area provided, and readied as the base address register.

DSECT=dsect_name

specifies the name of the dummy section with which the requested memory area is to be overlayed.

LENGTH=

specifies the length in bytes of the area to be requested directly via val, or via the contents of reg2.

INIT= specifies the name (int name) or address (in reg3) of an area in the same module (internal) whose data is to be copied for initialization in the requested memory area.

EXTINIT=ext_name

specifies the name of an area in any module (external) whose data is to be copied for initialization in the requested memory area.

Programming notes

The dummy section specified in DSECT=dsect_name must be terminated with the following assembler instruction:

```
Ldsect_name     EQU  *-dsect_name
```

Example

```
Name          | Operation  | Operands
--------------|------------|-----------------------------------------
EX1           | @ENTR      | TYP=E
              | @DATA      | CLASS=A,BASE=R7,DSECT=DATA,INIT=INDATA
              | .          |
              | .          |
              | @EXIT      |
              | .          |
              | .          |
INDATA        | DS         | ..                    (01)
              | .          |
              | .          |
              | @END       |
DATA          | DSECT      |                       (02)
..            | DS         | ..
              | .          |
              | .          |
LDATA         | EQU        | *-DATA                (03)
              | .          |
              | .          |
```

(01) Definition of data which is to be copied into the requested memory area.
(02) Definition of the dummy section which is to overlay the requested memory area.
(03) Termination of the dummy section, length calculation.

Format 2: Class S data areas.

Name	Operation	Operands
[name]	@DATA	CLASS=S ,BASE=reg [,INIT=int_name] {,EXTINIT=ext_name,DSECT=dsect_name}

name Name

reg General-purpose register; positive absolute expression, either
- decimal self-defining term or
- predefined name of a register
- name to to which an appropriate self-defining term has been allocated

int_name, ext_name
Names of data areas

dsect_name
Name of a dummy section

Description

CLASS=S
This is a static class data area.

BASE=reg
specifies the base address register to be used.

When @DATA is called, the register named is readied as the base address register and loaded with the start address of the data area.

INIT=int_name
specifies the name of a data area which is in the same module (internal).

EXTINIT=ext_name
specifies the name of a data area which is in any desired module (external).

DSECT=dsect_name
specifies the name of the dummy section which must describe the structure of the area ext_name.

Example

The example shows an access to data which is defined in an external module.

```
Name          | Operation | Operands                                              
--------------+-----------+-------------------------------------------------------
* Module A                                                                    (01)
              |     .     |
              |     .     |
              |   @DATA   | CLASS=S,BASE=R5,EXTINIT=BDATEN,DSECT=ADATEN
              |     .     |
              |     .     |
              |   @EXIT   |
              |   @END    |
ADATEN        |   DSECT   |                                                   (02)
A1            |   DS      | CL5
              |     .     |
              |     .     |
AEND          |   DS      | CL6
LADATEN       |   EQU     | *-ADATEN
* Module B                                                                   (03)
DATEN         |   START   |
              |   ENTRY   | BDATEN
BDATEN        |   DS      | 0CL100                                            (04)
B1            |   DC      | C'HALLO'
              |     .     |
              |     .     |
BEND          |   DC      | C'END  B'
```

(01) Module A contains the procedure which is to access the data.
(02) Dummy section which describes the structure of the external data area.
(03) Module B is the external module containing the data definition.
(04) Data definition in the external module.

Format 3: Class B data areas

Name	Operation	Operands
[name]	@DATA	CLASS=B ,BASE=reg ,DSECT=dsect_name

name Name

reg General-purpose register, positive absolute expression, either
- decimal self-defining term or
- predefined name of a register or
- name to which an appropriate self-defining term has been allocated

dsect_name
Name of a dummy section or a data area

Description

CLASS=B
This is a based class data area.

BASE=reg
specifies the base address register to be used.

reg must be loaded with the start address of the data area for which storage has already been reserved in another procedure.

DSECT=dsect_name
specifies the name of a data area which already exists or the name of a dummy section which describes the structure of the new data area.

Programming notes

In an earlier or dynamically higher-level procedure, reg must be loaded via a @DATA call with CLASS=A, C or S with the start address of the data area.

Example

```
Name          | Operation  | Operands
--------------+------------+-------------------------------------------
* Higher-level procedure
FIRST         | @ENTR      | TYP=M
              | @DATA      | CLASS=S,BASE=R9,INIT=CONST                (01)
              | .          |
              | .          |
              | @PASS      | NAME=SECOND                               (02)
              | @EXIT      |
              | @END       |
CONST         | DS         | 0D                                        (03)
              | DC         | ..
              | .          |
              | .          |

* Lower-level procedure
SECOND        | @ENTR      | TYP=I
              | @DATA      | CLASS=B,BASE=R9,DSECT=CONST               (04)
              | .          |
              | .          |
```

(01) Creation of the data area CONST with base address register R9.
(02) Call of the SECOND procedure.
(03) Definition of the data area CONST.
(04) Overlaying the new data area with the structure of the data area CONST.

@DO Loop sub-block

Function

@DO indicates the start of the loop sub-block in an iterative loop and in a loop with pre-check.

Format

Name	Operation	Operands
[name]	@DO	

>>>>> See also @BEND, @THRU and @WHIL

@ELSE NO sub-block

Function

@ELSE forms the heading of the NO sub-block in a decision.

Format

Name	Operation	Operands
[name]	@ELSE	

Description

The sub-block in an @IF branch which begins with @ELSE is only executed if the condition set with @IF does **not** hold true.

>>>>> See also @IF, @THEN and @BEND

@END Static procedure end

Function

@END denotes the static end of a procedure opened with @ENTR.

Format

Name	Operation	Operands
[name]	@END	[,LTORG={YES / NO}][,DROP={(reg[,...]) / ()}]

name Name

reg General-purpose register; positive absolute expression, either
- decimal self-defining term or
- predefined name of a register or
- name to which an appropriate self-defining term has been assigned

Description

Calling the @END macro results in
- generation of a LTORG instruction in all procedures, except type D. In other words, a literal pool is created starting on the next doubleword boundary (see 4.2, LTORG instruction).
- the release of all base address registers allocated using @ENTR and @DATA (see 4.2, DROP instruction).

LTORG=YES
Causes an LTORG instruction to be generated; default for all procedures, except type D.

LTORG=NO
No LTORG instruction is generated; default for type D procedures.

DROP=(reg[,...])
Besides the registers released conventionally, others may also be released.

DROP=() Generates a DROP instruction without operands, releasing all base registers previously defined with USING.

@ENTR Procedure start

Function

@ENTR forms the heading for all types of procedure. Depending on the type specified, various operands may be specified.

For all the following formats of @ENTR, the following general specifications are possible, in addition to the operands specified.

Format

Name	Operation	Operands
...	@ENTR	... [,VERS=xxx] [,AUTHOR=name] [,FUNCT='function'] [,CHECK={ON / OFF}] [,TITLE={YES / NO}]

xxx Version designation, specified unformatted.

name Name of the programmer.

function Documentation text; is to specify the function of the procedure; maximum length 63 characters.

Description

CHECK= determines the generation of checks at execution time with error recovery.

- ON Checks and error recovery are generated. The function is used for program security, at the expense of storage space and runtime.
- OFF Checks and error recovery are not created.

TITLE= controls the generation of a TITLE instruction.

- YES TITLE instruction is generated (default for all types of procedures, except D). The title contains the name of the procedure, its type, the version and the date of assembly.
- NO TITLE instruction is not generated (default for procedure type D).

Format 1: Main procedure

Name	Operation	Operands
name	@ENTR	TYP=M [,MAXPRM=val] [,LOCAL=dsect_name] [,AMODE={24 | 31 | ANY},RMODE={24 | ANY}] [,ENV=C[,LOADR12={YES | NO}]] [,STACK=n] [,ILCS={YES | NO}] [,STREQ=symb addr1] [,STREL=symb addr2] [,HPREQ=symb addr3] [,HPREL=symb addr4] [,SLTERM=symb addr5] [,SCTERM=symb addr6] [,EXTMIN=val1] [,ABKR={YES | NO}] [,PROCHK=symb addr1] [,ERROR=symb addr2] [,OTHEVT=symb addr3]

name Name of the main procedure

val Decimal self-defining term; number of parameters to be passed

dsect_name
Name of a data area

symb addr1...symb addr6
Symbolic addresses of the user-own routines

val1 Decimal self-defining term that specifies the minimum stack extent size in bytes.

symb addr1...symb addr3
Symbolic addresses of handling routines

Description

TYP=M Specifies that this is the main procedure of a program.

If several modules are combined into an executable program, only one module may contain a TYP=M procedure.

MAXPRM=val
Must be specified for dynamic parameter passing via the STANDARD interface. val specifies the maximum number of parameters which are to be passed to the called procedure with the @PASS macro (see @PASS, format 3).

The storage space for the parameter is reserved in the procedure STACK. The MAXPRM= specification is ignored if the operand LOCAL= is set at the same time, as the LOCAL area and the area for the parameter overlap in the STACK procedure.

LOCAL=dsect_name
Must be specified if a local data area is to be reserved in the STACK procedure.

Register 13 is used as the base address register for addressing this area.

The structure of the area requested within the STACK procedure is described by a dummy section, which must be defined with dsect_name @PAR (see @PAR, format 3).

AMODE= Assigns an addressing mode to the procedure (see 4.2, AMODE instruction).

RMODE= Assigns a load attribute to the procedure (see 4.2, RMODE instruction).

If there is an invalid combination of AMODE and RMODE, an MNOTE is generated, and AMODE 24 and RMODE 24 are entered for both values.

ENV=C Only allowed with ILCS=NO
Must be specified if the procedure concerned is to behave as a C program. In other words, the procedure runs under the control of the C runtime system (see "ASSEMBH User Guide" [1]).

LOADR12=YES
The address of the program manager for C programs (see "C Compiler User Guide" [9]) is to be loaded into register 12.

LOADR12=NO
Register 12 is not loaded.

Register 12 should already contain the address of the program manager. This is always the case if the calling procedure is a C program or if register 12 was not changed in an assembly language program with ENV=C.

STACK=n Size of the dynamically extendable initial stack in bytes.

If no entry is made, an initial stack of one page (= 4096 bytes) is requested.

ILCS=YES Connection to ILCS

ILCS=NO Default value
Non-ILCS procedure

ILCS allows the user to enable user-own routines for reserving and releasing memory and for termination handling at initialization time. These routines must comply with ILCS conventions. The minimum stack extent size can also be specified. These specifications are valid throughout the entire ILCS environment and can be defined only in the main procedure @ENTR with TYP = M and ILCS=YES, using the following operands:

STREQ=

symb addr1
Symbolic address of the user-own memory reservation routine for stack management

Input parameter:	byte length of the memory area, with alignment on a doubleword boundary
Return value in R0:	pointer to memory area
Return code in R15:	=F’0’, no errors occurred

not specified
Memory is reserved in ILCS via REQM

STREL=

symb addr2
: Symbolic address of the user-own memory release routine for stack management
  Input parameter: pointer to memory area
  byte length of the memory area
  Return code in R15: =F'0', no errors occurred

not specified
: Memory is released in ILCS via RELM

HPREQ=

symb addr3
: Symbolic address of the user-own memory reservation routine for heap management
  Input parameter: byte length of the memory area, with alignment on a doubleword boundary
  Return value in R0: pointer to memory area
  Return code in R15: =F'0', no errors occurred

not specified
: Memory reserved in ILCS via REQM

HPREL=

symb addr4
: Symbolic address of the user-owm memory reservation routine for heap management
  Input parameter: pointer to memory area
  byte length of the memory area
  Return code in R15: =F'0', no errors occurred

not specified
: Memory is released in ILCS via RELM

SLTERM=

symb addr5
: Symbolic address of the user-own termination routine

not specified
: No user-own termination routine

SCTERM=

symb addr6
: Symbolic address of the user-own termination routine for STXIT and contingency processes
in R15 error code: >0 internal error
in R0 and R1 information code:
= 0: it is an STXIT process
= 1: it is a contingency process

not specified
: If internal errors occur, STXIT or contingency process is aborted with TERM UNIT=STEP

EXTMIN=

val1
: Decimal self-defining term that specifies the minimum stack extent size in bytes. The specification is rounded up to a number of pages equal to the next power of 2 (4096 bytes).

not specified
: Minimum stack extent size is 16 pages.

With the standard event handler (SEH), ILCS provides a routine for coordinating events that occur in ILCS programs (see also section 9.6.3, "Event handling"). Assuming ILCS=YES has been set, the following operands are possible:

ABKR= Set/do not set abort identifier

YES
: An identifier for the standard event handler is entered in the appropriate event handler list (EHL) of the procedure in order to abort the search for handling routines within the save area chain.

NO
: The abort identifier is not set.

PROCHK=

symb addr1
: Symbolic address of a handling routine for STXIT events of class ’PROCHK’.

not specified
: No handling routine for class ’PROCHK’ is made available to the standard event handler.

ERROR=

symb addr2
: Symbolic address of a handling routine for STXIT events of class ’ERROR’.

not specified
: No handling routine for class ’ERROR’ is made available to the standard event handler.

OTHEVT=

symb addr3
: Symbolic address of a handling routine for non-STXIT events.

not specified
: No handling routine for class ’OTHEVT’ is made available to the standard event handler.

Programming notes

1. The abort identifier is required in the event handler list if the standard event handler has to abort the search for event handling routines because of a switch from an ILCS procedure to a non-ILCS procedure within the procedure nesting structure.
2. If you use TEST-SUPPORT=YES when assembling, there must on no account be a CSECT instruction with a name differing from the @ENTR name entry preceding the invocation of @ENTR Typ=M, because this CSECT is terminated by the consistency constant generated for AID. This may result in undefined program behavior if the program section is not exited before the constant.

Format 2: Type I or E procedures

Name	Operation	Operands
name1	@ENTR	TYP={I / E} [,PLIST=({name2 / (reg)}[,...])] [,PASS={OPT[IMAL] / STA[NDARD]}] [,MAXPRM=val] [,LOCAL=dsect_name] [,RETURNS={YES / NO}] [,ENTRY={ENTRY / CSECT[,AMODE={24 / 31 / ANY},RMODE={24 / ANY}]}] [,ENV=C[,LOADR12={YES / NO}]] [,ILCS={YES / NO}] [,ABKR={YES / NO}] [,PROCHK=symb addr1] [,ERROR=symb addr2] [,OTHEVT=symb addr3]

name1 Procedure name
name2 Formal parameter
reg Register used as a formal parameter; positive absolute expression, either
- decimal self-defining term or
- predefined name of a register or
- name to which an appropriate self-defining term has been assigned

val Decimal self-defining term; number of parameters to be passed
dsect_name
Name of a data area

symb addr1...symb addr3
Symbolic addresses of handling routines

Description

TYP=I Specifies that this is a procedure which can only be called from the module in which it lies (internal procedure) and which is connected to memory management and register saving.

TYP=E Specifies that this is a procedure which may be called from any module (external procedure) and which is connected to memory management and register saving.

PLIST= is specified for the acceptance of parameters.

The operand specifies the formal parameters, a list of data fields and registers into which entries may be accepted when the procedure is called (see @PAR, format 2).

For parameter acceptance into the LOCAL area of the procedure, the PLIST operands must match the PLIST operands of the associated @PAR.

PASS=OPT
Only allowed with ILCS=NO
Must be specified if parameters are to be passed in the called procedure via the OPTIMAL interface.

Registers 1 to 4 are used for parameter passing.

PASS=STA Parameters are passed in the called procedure via the STANDARD interface.

Only register 1 is used for acceptance of parameters. Register 1 contains the address of the parameter list.

The PASS= specification must match the PASS operand in the @PASS macro of the calling procedure (see @PASS, format 3).

MAXPRM=val

Must be specified for dynamic parameter passing via the STANDARD interface. val specifies the maximum number of parameters which can be transferred to a procedure (see @PASS, format 3).

LOCAL=dsect_name

Must be specified if a local data area is to be reserved in a STACK procedure.

The base address register for this area is register 13.

The structure of the requested area is described by a dummy section, which must be defined with dsect_name @PAR. dsect_name may

- denote a @PAR which defines a local dummy section (format 3 of @PAR) or
- denote a @PAR which defines a dummy section for parameter acceptance (format 2 of @PAR).

RETURNS=

Regulates which registers are reloaded with the original values (before calling the procedure) on quitting the procedure via @EXIT, and whether a function value is returned.

The operand ILCS=YES/NO has an influence on the way in which RETURNS=YES/NO operates.

YES If ILCS=NO:
Registers 0 and 2 to 14 are reloaded, register 1 is not. This means that register 1 can be used to send a function value to the calling procedure. The called procedure thus becomes a "function procedure".

If ILCS=YES:
The procedure returns a function value. Registers 0 through 14 are reloaded. The function value in register 1 is copied to register 0 in the procedure epilog if the caller is an ILCS procedure.

NO If ILCS=NO:
All registers 0 through 14 are reloaded.

If ILCS=YES:
The procedure does not return a function value, and thus R1 is not copied to R0 in the procedure epilog. The register contents of R0 and R1 are undefined. Registers 2 through 14 are reloaded.

not specified
Registers 2 to 14 are reloaded.

When ILCS=NO, the reloading of the registers can also be restricted using the @EXIT macro operand RESTORE=MIN.

ENTRY= Specifies which input instructions are to be generated for the procedure.

ENTRY= may only be specified in type E procedures.

ENTRY, if specified, generates:

```
name1  DS     0D
ENTRY  name1
```

This specification is mandatory if the AID commands %CONTROL or %TRACE are to be used for debugging a program with more than one procedure (see the manual "AID - Debugging of ASSEMBH Programs" [2]).

CSECT, if specified generates:

```
name1    CSECT
```

AMODE= Assigns an addressing mode to the procedure (see 4.2, AMODE instruction).

RMODE= Assigns a load attribute to the procedure (see 4.2, RMODE instruction).

If there is an invalid combination of AMODE and RMODE, an MNOTE is generated, and AMODE 24 and RMODE 24 are entered for both values.

ENV=C Must be specified if the procedure concerned is to behave like a C program; that is, the procedure runs under the control of the C runtime system (see "ASSEMBH User Guide" [1]).

It may only be specified for type E procedures with ILCS=NO.

LOADR12=YES
The address of the program manager for C programs (see "C Compiler User Guide" [9]) is to be loaded in register 12.

LOADR12=NO
: Register 12 is not loaded.

  Register 12 should already contain the address of the program manager. This is always the case if the calling procedure is a C program or if register 12 was not changed in an assembly language program with "ENV=C".

ILCS=YES Connection to ILCS

ILCS=NO Default value
Non-ILCS procedure

With the standard event handler (SEH), ILCS provides a routine for coordinating events that occur in ILCS programs (see also section 9.6.3, "Event handling").
If ILCS=YES has been set, the following operands are possible:

ABKR= Set/do not set abort identifier

YES An identifier for the standard event handler is entered in the appropriate event handler list (EHL) of the procedure in order to abort the search for handling routines within the save area chain.

NO The abort identifier is not set.

PROCHK=

symb addr1
: Symbolic address of a handling routine for STXIT events of class ’PROCHK’.

not specified
: No handling routine for class ’PROCHK’ is made available to the standard event handler.

ERROR=

symb addr2
: Symbolic address of a handling routine for STXIT events of class ’ERROR’.

not specified
: No handling routine for class ’ERROR’ is made available to the standard event handler.

OTHEVT=

symb addr3
: Symbolic address of a handling routine for non-STXIT events.

not specified
: No handling routine for class ’OTHEVT’ is made available to the standard event handler.

Programming notes

The abort identifier is required in the event handler list if the standard event handler has to abort the search for event handling routines because of a switch from an ILCS procedure to a non-ILCS procedure within the procedure nesting structure.

Example of format 1 and format 2

```
Name        | Operation | Operands
------------------------------------------------------------------
* Calling procedure
MAIN        | @ENTR     | TYP=M,MAXPRM=2,LOCAL=DUMMY                 (01)
            | .         |
            | .         |
            | @PASS     | NAME=PRO2,PLIST=(FIELD1,FIELD2)            (02)
            | .         |
            | .         |
DUMMY       | @PAR      | D=YES                                      ]
NAME1       | DS        | ..                                         |
            | .         |                                            } (03)
            | .         |                                            |
DUMMY       | @PAR      | LEND=YES                                   ]
            | .         |
            | .         |
            |           |

* Called procedure
PRO2        | @ENTR     | TYP=I,LOCAL=IN,PLIST=(INFIELD1,INFIELD2)   (04)
            | .         |
            | .         |
IN          | @PAR      | D=YES,LEND=YES,PLIST=(INFIELD1,INFIELD2)   (05)
            | .         |
            | .         |
```

(01) The procedure MAIN
- is to pass a maximum of 2 parameters to another procedure (MAXPRM=2) and
- a local data area is requested for it. The structure of the local data area is described in the dummy section, DUMMY.

(02) Calling PRO2, 2 parameters are to be passed to PRO2; they are passed dynamically via the STANDARD interface.

(03) Definition of the dummy section DUMMY.

(04) PRO2 accepts 2 parameters; the dummy section IN describes the structure of the data area readied for the acceptance; PLIST=... specifies the parameter list.

(05) Definition of the dummy section IN.

Format 3: Type B or L procedures

Name	Operation	Operands
name	@ENTR	TYP={B / L} [,BASE=reg] [,LOADSB={YES / NO}] [,ENTRY={ENTRY / CSECT[,AMODE={24 / 31 / ANY},RMODE={24 / ANY}]}]

name Procedure name

reg General-purpose register; positive absolute expression, either
- decimal self-defining term or
- predefined name of a register or
- name to which an appropriate self-defining term has been allocated

Description

TYP=B specifies that this is a procedure which may be called from any module (external procedure), and which is not connected to memory management and register saving.

TYP=L specifies that this is a procedure which may only be called from the same module (internal procedure), and which is not connected to memory management and register saving.

BASE=reg allocates a register to the procedure as the base address register.

If this is not specified, the procedure is assigned register 15 as the base address register by the assembler.

LOADSB=

regulates loading of the base address register after each call of another sub-procedure with @PASS.

YES The base address register specified with the BASE operand or the standard base address register 15 is loaded with the address of the calling procedure.

NO No register is loaded after the call, not even register 15.

not specified
Register 15 is loaded with the address of the calling procedure after the call.

ENTRY= specifies which entry instruction statements are to be generated for the procedure.

ENTRY= may only be specified in type B procedures.

If specified, ENTRY generates:

```
name1  DS     0D
ENTRY  name1
```

If specified, CSECT generates:

```
name1    CSECT
```

AMODE= assigns an addressing mode to the procedure (see 4.2, AMODE instruction).

RMODE= assigns a load attribute to the procedure (see 4.2, RMODE instruction).

If there is an invalid combination of AMODE and RMODE, a MNOTE will be generated, and AMODE 24 and RMODE 24 are entered for both values.

Programming notes

1. A base address register, assigned with BASE=reg, must be loaded explicitly at the start of the procedure (after the @ENTR macro), e.g. by means of the machine instruction: `LR  reg,R15`
2. This procedure type is only allowed when ILCS=NO is specified.

Format 4: Type D procedures

Name	Operation	Operands
[name]	@ENTR	TYP=D

name Procedure name

Description

This is a type D procedure, without connection to memory management and register saving. The procedure may be external or internal.

Programming notes

1. If the procedure is the first in a module, no procedure name may be specified. The procedure receives the name of the START or CSECT instruction.
2. This procedure type is only allowed when ILCS=NO.

@EVTLC Define event layout context

This macro instruction is only allowed in procedures with ILCS=YES.

Function

@EVTLC (**Ev**en**t L**ayout **C**ontext) defines the layout of the context description for the @EVTOE macro.

Format

Name	Operation	Operands
[name]	@EVTLC	[pre]=string

name Name
pre Prefix
string Alphanumeric characters

Description

pre Prefix for the symbolic names of individual fields

= string No more than 4 characters
pre must conform to the assembler syntax for names.

The prefix is used to define the following DS statements for the individual fields of the context description:

```
preCR0     DS    F        Context registers
preCR1     DS    F
preCR2     DS    F
preCR3     DS    F
preCR4     DS    F
preCR5     DS    F
preCR6     DS    F
preCR7     DS    F
preCR8     DS    F
preCR9     DS    F
preCR10    DS    F
preCR11    DS    F
preCR12    DS    F
preCR13    DS    F
preCR14    DS    F
preCR15    DS    F
preCPC     DS    F        Program counter
preCEVC    DS    F        Event code
preCFP0    DS    2F       Floating-point registers
preCFP2    DS    2F
preCFP4    DS    2F
preCFP6    DS    2F
preCILC    DS    X        Instruction length code
preCCC     DS    X        Condition code
preCPM     DS    X        Program mask
preCSS     DS    X        Language key
```

Programming notes

A prefix with more than 4 characters is truncated to 4 characters.

@EVTOE Signal non-STXIT event

This macro instruction is only allowed in procedures with ILCS=YES.

Function

@EVTOE (**Ev**en**t**ing **o**ther **e**vent) signals a non-STXIT event (see also section 9.6.3, "Event handling").

Format

Name	Operation	Operands
[name]	@EVTOE	CONTXT={symb addr / (reg)}

name Name
symb addr Symbolic address of a data area
reg General-purpose register containing the address of a data area

Description

CONTXT= Describes the environment of the event location.

symb addr
Symbolic address of a data area 112 bytes in length. The user must store in this area the context of the program at the point where the event occurred. This includes the contents of all standard (context) and floating-point registers, the program counter, the program mask, and the user-defined event code for the handling routine called by ILCS.

(reg) Optionally, a register containing the address of the data area.

See the section on the @EVTLC macro for the structure of this data area.

The inline code of the macro copies the specifications from the macro into an ILCS-conforming parameter block; these specifications are then used to activate the corresponding entry in the standard event handler in order to signal a non-STXIT event.

When the standard event handler returns control to the calling procedure, R15 contains a return code which indicates whether or not the function has been executed, and which errors, if any, occurred.

Programming notes

Individual context fields cannot be checked; the user is therefore responsible for ensuring the accuracy and consistency of context data.

@EXIT Dynamic procedure end

Function

@EXIT terminates the called procedure and returns control to the calling procedure.

Format 1: Return from type M, B or L procedures

Name	Operation	Operands
[name]	@EXIT	[LOADR12={YES / NO}]

Description

In type M procedures, @EXIT terminates the program.

In type B or L procedures, the program is continued with the instruction statement which follows the @PASS macro in the calling procedure.

LOADR12=

Can only be specified for procedures in which "ENV=C" and ILCS=NO are set.

YES The address of the program manager for C programs (see "C Compiler User Guide" [9]) should be entered in register 12.

NO Register 12 is not loaded.

Register 12 should already contain the address of the program manager. This is always the case if the calling procedure is a C program or if register 12 was not changed in an assembly language program with "ENV=C".

Format 2: Return from type E or I procedures

Name	Operation	Operands
[name1]	@EXIT	[{RC={name2 | (reg) | val} | TO=proc_name | RESTORE=MIN | PROG=FORTRAN | LOADR12={YES | NO}}][,...]

name1 Name
name2 Name of a data field which contains the return value
reg General-purpose register which contains the return value or the address of the return value; positive absolute expression, either
- decimal self-defining term or
- predefined name of a register or
- name to which an appropriate self-defining term has been allocated

val Self-defining term which specifies the return value directly
proc_name Name of a procedure which must be higher in the call hierarchy than the calling procedure.

Description

After @EXIT, the program is continued with the instruction statement following the @PASS macro in the calling procedure.

Using @EXIT, registers 2 to 14 are reloaded with their values during the procedure call.

@EXIT macro instruction in an ILCS procedure (@ENTR ILCS=YES)

If the @ENTR parameter RETURNS=YES was set, the function value in register 1 is copied to register 0. If RETURNS=NO was set, the function value is not copied.

RC= Only to be specified in procedures in which ILCS=NO is set.
Specifies a return value which was calculated in the called procedure and is to be passed to the calling procedure. The value is passed in register 15 by default, or in register 0 in the case of non-ILCS FORTRAN programs.
If you specify val, the value is passed; if you specify name2, the address is passed.
If you specify (reg), the register contents are transferred to the return value register.

TO=proc_name

can only be specified in procedures in which ILCS=NO is set.
This must be specified if there is to be no return to the calling procedure but to a higher-level one in the call hierarchy.

The program is continued with the instruction statement which follows the actual @PASS macro in the procedure identified by proc_name.

RESTORE=MIN

Is specified if, on return to the calling procedure, only registers 7 to 14 are to be reloaded.

PROG=FORTRAN

Can only be specified in procedures in which ILCS=NO is set.
Must be specified if a return is to be made from a called assembler procedure to a calling FORTRAN program. In this case, registers 2 to 14 are reloaded.

LOADR12=

Can only be specified for procedures in which ENV=C and ILCS=NO are set.

YES The address of the program manager for C programs (see "C Compiler User Guide" [9]) is to be loaded into register 12.

NO Register 12 is not loaded.

Register 12 should already contain the address of the program manager. This is always the case if the calling program is a C program or if register 12 was not changed in an assembler language program with "ENV=C".

Programming notes

The @EXIT operand RC=<return code> causes the return code to be passed in register 15. Within structured assembler programs, register 15 can be evaluated by the user in the normal way (ILCS register conventions require only registers 2 through 14 to be reloaded in the procedure epilog). However, calling procedures in other environments generally cannot then interrogate this return code since, according to ILCS conventions, register 15 is regarded as having been destroyed. The @EXIT RC operand should therefore not be used if this can be avoided and the return code should instead be passed as a function value in register 1 or as an output parameter.

Example

```
Name      | Operation | Operands
----------+-----------+-------------------------------------
PRO1      | @ENTR     | TYP=M
          | @DATA     | CLASS=S,BASE=R9,INIT=CONST
          | @PASS     | NAME=PRO2                      (01)
          | @IF       | EQ                            ⎫
          | CLI       | TEST,C'A'                     ⎪
          | @THEN     |                               ⎬ (02)
          | .         |                               ⎪
          | .         |                               ⎭
          | @BEND     |
          | .         |
          | .         |
          | @EXIT     |
          | @END      |
CONST     | DS        | 0H
TEST      | DS        | CL1                            (03)
          | .         |
          | .         |
PRO2      | @ENTR     | TYP=I
          | @DATA     | CLASS=B,BASE=R9,DSECT=CONST
          | .         |                               ⎫
          | .         |                               ⎬ (04)
          | MVI       | TEST,C'A'                     ⎭
          | @EXIT     | RC=TEST                        (05)
          | @END      |
```

(01) Calling the PRO2 procedure.
(02) Depending on the return value, instruction statements are to be executed in PRO1.
(03) Definition of the field which contains the return value.
(04) The return value must be provided in PRO2.
(05) Return to PRO1; the name of the field containing the return value is transferred.

The same function could have been fulfilled by means of the following instruction statements in PRO2:

```
LA        | R8,TEST
@EXIT     | RC=(R8)
```

@FREE Memory release

Function

@FREE causes the release of the memory area earlier requested via @DATA CLASS=C.

Format

Name	Operation	Operands
[name]	@FREE	BASE=reg

name Name
reg General-purpose register, positive absolute expression, either
- decimal self-defining term or
- predefined name of a register or
- name to which an appropriate self-defining term has been allocated

Description

reg specifies the address of the memory area to be released, and must match the BASE operand of the relevant @DATA macro.

>>>>> See also @DATA

@IF Decision

Function

@IF forms the heading of a branch in which one of two alternatives must be selected.

Format

Name	Operation	Operands
[name]	@IF	cond_sym

name Name
cond_sym Predefined or user-own condition symbol (see sections 9.2.4 and 9.2.5).

Description

cond_sym specifies the condition (see sections 9.2.4 and 9.2.5) that is to be set for the branch.

If the condition holds true, the @THEN branch of the structure block is executed, if not, the @ELSE branch is.

Example

Name	Operation	Operands
	@IF	LE
	CR	R1,R2
	@THEN	
	MVI	FIELD,TRUE
	@ELSE	
	MVI	FIELD,FALSE
	@BEND	

If the contents of register 1 are less than or equal to those of register 2, the @THEN sub-block is executed, otherwise the @ELSE sub-block is.

>>>>> See also @THEN, @ELSE and @BEND

@ININ Call ILCS for dynamically loaded modules

This macro instruction is allowed only in an ILCS environment. ILCS must have been initiated at an earlier point.

Function

@ININ calls ILCS to handle retroactive initialization of a runtime system linked in by large, dynamically loaded modules (see also section 9.6).

Format

Name	Operation	Operands
[name]	@ININ	

name Name

Description

The inline code of the macro generates the activation of the ILCS routine in order to check and, if appropriate, perform language initialization.

Following the macro call, register 15 contains a return code which indicates whether or not the function has been executed.

Programming notes

An error occurring during the initialization check can result in undefined program behavior (e.g. memory bottleneck); this causes the ILCS termination routine to be called. If this happens, all languages initialized in the ILCS environment will also be terminated.

@OF Case sub-block

Function

@OF forms the heading of a sub-block in a case differentiation by comparison, and specifies it or the actual comparand.

Format

Name	Operation	Operands
[name1]	@OF	{name2 / literal / val}[,...][,COMP=instr]

name1 Name
name2 Name of the field containing a comparand
literal Literal which specifies a comparand directly (see section 2.5.3)
val Self-defining term which specifies a comparand directly
instr Machine instruction which sets a condition code (see the "Assembler Instructions" reference manual).

Description

The sub-block which begins with @OF and the actual comparand, is executed if one of the comparands specified matches the selector from the @CAS2 macro.

Using COMP=instr, a non-standard comparison type may be selected for this sub-block.

Programming notes

1. The user must ensure that selector and comparand correspond as regards length, alignment and compare instruction.
2. A self-defining term may only be specified as a comparand if a machine instruction which permits a direct operand as the second operand was specified with COMP=instr in the @OF macro or in the @CAS2 macro.

Example

```
Name        | Operation | Operands
------------+-----------+-------------------
            | @CAS2     | FIELDA              (01)
            | @OF       | FIELDB              (02)
            | .         |
            | .         |
            | @OF       | C'*',COMP=CLI       (03)
            | .         |
            | .         |
```

(01) Start of case differentiation; FIELDA is the name of the field which contains the selector.

(02) Start of the first sub-block; this is executed if FIELDA is equal to FIELDB.

(03) Start of the second sub-block; this is executed if FIELDA coincides with *. The compare instruction for this sub-block only is CLI.

>>>>> See also @CAS2, @OFRE and @BEND

@OFRE Remainder sub-block

Function

In case differentiation by comparison, @OFRE forms the heading of the last sub-block

Format

Name	Operation	Operands
[name]	@OFRE[ST]	

Programming notes

In this sub-block all conditions are dealt with that did not occur in the other sub-blocks. This block should be used for error conditions, or for conditions that are irrelevant for case differentiation.

>>>>> See also @CAS2 and @OF

@OR Logical 'OR'

Function

@OR implements the logical OR operation in compound conditions.

Format

Name	Operation	Operands
[name]	@OR	cond_sym

name Name
cond_sym Predefined or user-own condition symbol (see sections 9.2.4 and 9.2.5).

Programming notes

The call of the @OR macro must be followed by a condition code setting machine instruction (see "Machine Instructions" manual).

Example

```
  Name        Operation   Operands
              .
              .
              @IF         ZE
              C           R4,ZERO             (01)
              @OR         NZ
              LTR         R5,R5               (02)
              @THEN
              ST          R1,POINT
              @ELSE
              .
              .
              @BEND
              .
              .
```

The @THEN branch is executed if the *first condition* (01) or *second condition* (02) holds true.

>>>>> See also @AND and @TOR

@PAR Definition of areas

Function

@PAR has three different functions. Depending on the format, a parameter list is created or a dummy section defined for parameter passing, or a dummy section is defined for parameter acceptance.

Format 1: Parameter list for static parameter passing via the STANDARD interface

Name	Operation	Operands
list_name	@PAR	[PLIST=({name1 / (val1)}[,...]),] VLIST=({name2 / (val2)}[,...]) [,PLEND={YES / NO}]

list_name Name of the parameter list (63 characters)
name1 Name of the entries in the parameter list
val1 Decimal self-defining term; generates an unnamed entry in the parameter list
name2 Name of the fields whose addresses are to be transferred to the parameter list (actual parameters)
val2 Decimal self-defining term; is entered directly in the parameter list (actual parameter)

Description

Calling @PAR causes a parameter list with constant values to be created at assembly time.

PLIST= gives names to the parameter list entries.

A nameless address constant is created by specifying (val1).

VLIST= specifies the actual parameters whose addresses or values are included in the parameter list.

If no actual parameter is assigned to a parameter, an extra comma must appear at this point in the VLIST operand. This type of specification is treated in the same way as the self-defining term 0.

PLEND= Can only be specified in procedures in which ILCS=YES is set. Specifies whether the most significant bit is set for the last parameter when the static parameter address list is set up.

YES The bit is set

NO The bit is not set

not specified:
If ILCS = NO, the bit is set for compatibility reasons;
if ILCS = YES, the bit is not set.

The actual parameters must be passed using "call by reference". Self-defining parameter values may not be passed in the actual parameter list (unless absolute addresses are involved). The list may contain empty list elements, but these should be regarded only as "placeholders". Before calling a procedure in which the list is passed (@PASS PAR=<par-list>), the user must fill the places left free in it with correct parameter address values ("call by reference").

Programming notes

1. No names from dummy sections (see section 4.2, DSECT instruction) may be specified in the VLIST operand.
2. The most significant bit of the last address constant is set to 1 to denote the end of the parameter list. Negative values should therefore never be transferred directly.
3. @PAR must be called between @EXIT and @END of the appropriate procedure.
4. @PAR macro instruction in an ILCS procedure (@ENTR ILCS=YES):
   - The number of parameters is stored in a new EQU constant. The name of the EQU constant is derived from the name of the parameter address list plus an additional suffix character "#".
   - The name of the parameter list may not be more than 63 characters, since the name of the EQU constant to be generated is exactly one character longer.

Example

```
Name        | Operation | Operands
------------|-----------|-----------------------------------------------
            | @PASS     | NAME=PROX,PAR=PARLIST
            | .         |
            | .         |
            | @EXIT     |
PARLIST     | @PAR      | PLIST=((1),A,B,C,D),VLIST=(ADR,4,NAME,,FIELD)
            | .         |
            | .         |
```

```
* Generated instruction statements
PARLIST     | DS        | 0F
            | DC        | A(ADR)
A           | DC        | A(4)
B           | DC        | A(NAME)
C           | DC        | A(0)
D           | DC        | A(FIELD+X'80000000')
```

>>>>> See also @PASS, format 2

Format 2: Definition of a dummy section in the LOCAL area for parameter acceptance

Name	Operation	Operands
dsect_name	@PAR	D=YES, PLIST=({ name / (reg) } [,...]), LEND=YES

dsect_name
: Name of the dummy section to be generated

name
: Formal parameter; name of a field in which the corresponding parameter is accepted when the call is made.

reg
: Register in which the corresponding parameter is accepted when the call is made; positive absolute expression, either
  - decimal self-defining term or
  - predefined name of a register or
  - name to which an appropriate self-defining term has been allocated

Description

Calling @PAR causes a dummy section to be generated in the called procedure. This specifies the list of formal parameters, i.e. it describes, or redefines, the structure of the storage space of the LOCAL area in the procedure STACK of the called procedure.

A separate entry is generated in the dummy section for each formal parameter name from the PLIST operand.

Format 2 of @PAR is allowed only in type E or I procedures.

Programming notes

1. The PLIST operands of @PAR must match the PLIST operands of the corresponding @ENTR.
2. The @PAR macro must follow the appropriate procedure after the @END call.

3. If the dummy section contains data other than the list of fields and registers to be accepted, it must be described as follows:

```
Name          | Operation | Operands
--------------|-----------|-----------------------------------------
              | .         |
              | .         |
              |           |                    ⎧name ⎫
dsect_name    | @PAR      | D=YES,PLIST=(      ⎨     ⎬[,...])
              |           |                    ⎩(reg)⎭...
              | DS        | ...
              | .         |
              | .         |
dsect_name    | @PAR      | LEND=YES
```

Example

The example shows the generation of a DSECT in the LOCAL area of a called procedure.

```
Name          | Operation | Operands
--------------|-----------|-----------------------------------------------
PRO           | @ENTR     | TYP=E,LOCAL=IN,PLIST=(INPAR1,INPAR2,(R5),INPAR3)
              | .         |
              | .         |
              | @END      |
IN            | @PAR      | D=YES,LEND=YES,PLIST=(INPAR1,INPAR2,(R5),INPAR3)
              | .         |
              | .         |

* Generated instruction statements
IN            | DSECT     |
              | ORG       | *+96
INPAR1        | DS        | A
INPAR2        | DS        | A
INPAR3        | DS        | A
LIN           | EQU       | *-IN
PRO           | CSECT     |
```

>>>>> See also @ENTR and @PASS, format 3

Format 3: Definition of the dummy section for the LOCAL area

Name	Operation	Operands
dsect_name	@PAR	{D=YES | LEND=YES}

dsect_name
Name of the dummy section to be generated

Description

D=YES @PAR with this operand denotes the start of the dummy section.

LEND=YES
@PAR with this operand denotes the end of the dummy section.

Data fields must be defined between @PAR D=YES and @PAR LEND=YES. This string of instruction statements can be used to describe the structure of a local data area in the procedure STACK, which was requested via @ENTR..., LOCAL=dsect_name.

Programming notes

The name of the dummy section to be generated must match the LOCAL operand of @ENTR in the corresponding procedure.

Example

```
Name      Operation   Operands
PRO       @ENTR       TYP=I,LOCAL=DUMMY
          .
          .
          @END
DUMMY     @PAR        D=YES
NAME1     DS          ..
          .
          .
DUMMY     @PAR        LEND=YES
```

>>>>> See also @ENTR

@PASS Procedure call

Function

@PASS implements a sub-procedure call. Here, the calling procedure can pass parameters to the called procedure.

Format 1: Call without parameter passing

Name	Operation	Operands
[name1]	@PASS	[NAME=int_name] [EXTNAME=ext_name] {[ADDR=[name2]] [ADDR=[(reg)]]}

name1 Name

int_name Name of a procedure in the same module as the calling procedure (internal procedure)

ext_name Name of a procedure in a module other than the calling procedure (external procedure)

name2 Name of a word which contains the address of the called procedure

reg Register which contains the address of the called procedure; positive absolute expression; either

- decimal self-defining term or
- predefined name of a register or
- name to which an appropriate self-defining term has been allocated

Programming notes

1. A procedure called with EXTNAME=ext_name must be identified with @ENTR TYP=E.
2. When specifying ADDR=, name2 or reg must be entered in the procedure, along with the address of the called procedure.

Format 2: Call with static parameter passing via the STANDARD interface

Name	Operation	Operands
[name1]	@PASS	{NAME=int_name / EXTNAME=ext_name / ADDR={name2 / (reg)}},PAR={list_name / (list_reg)}

name1 Name

int_name Name of a procedure in the same module as the calling procedure (internal procedure)

ext_name Name of a procedure in a module other than the calling procedure (external procedure)

name2 Name of a word containing the address of the called procedure

reg Register containing the address of the called procedure; positive absolute expression; either
- decimal self-defining term or
- predefined name of a register or
- name to which an appropriate self-defining term has been allocated

list_name Name of the parameter list

list_reg Only allowed in procedures in which ILCS=NO is set.
Register which must contain the parameter list address; positive absolute expression, either
- decimal self-defining term or
- predefined name of a register or
- name to which an appropriate self-defining term has been allocated.

Description

list_name or list_reg specifies the name, or the address, of the parameter list generated by the @PAR macro in the calling procedure.

The parameter list address is passed via register 1 to the called procedure.

If ILCS=YES is set, register 0 is loaded with the number of parameters.

Register 1 may not be used in the ADDR=reg operand.

Example

```
Name      | Operation | Operands
----------|-----------|-----------------------------------------------
PRO1      | @ENTR     | TYP=M
          | .         |
          | .         |
          | @PASS     | NAME=PRO2,PAR=PARLIST
          | .         |
          | .         |
          | @EXIT     |
PARLIST   | @PAR      | PLIST=(PAR1,PAR2,PAR3),VLIST=(FIELD,27,SET)
          | @END      |
          | .         |
          | .         |
PRO2      | @ENTR     | TYP=I,LOCAL=IN,PLIST=(INPAR1,INPAR2,INPAR3)
          | .         |
          | .         |
          | @EXIT     |
IN        | @PAR      | D=YES,LEND=YES,PLIST=(INPAR1,INPAR2,INPAR3)
          | .         |
          | .         |
```

>>>>> See also @PAR, format 1

Format 3: Call with dynamic parameter passing, STANDARD or OPTIMAL interface

Name	Operation	Operands
[name1]	@PASS	{NAME=int_name / EXTNAME=ext_name / ADDR={name2 / (reg)}} ,PLIST=({par_name / (par_reg)}[,...]) [,PASS={STA[NDARD] / OPT[IMAL]}] [,PLEND={YES / NO}]

name1 Name

int_name Name of a procedure in the same module as the calling procedure (internal procedure)

ext_name Name of a procedure in a module other than the calling procedure (external procedure)

name2 Name of a word which contains the address of the called procedure

reg Register which contains the address of the called procedure; positive absolute expression; either
- decimal self-defining term or
- predefined name of a register or
- name to which an appropriate self-defining term has been allocated

par_name Name of the field whose address is to be transferred to the called procedure

par_reg Register whose contents are to be transferred to the called procedure; positive absolute expression; either
- decimal self-defining term or
- predefined name of a register or
- name to which an appropriate self-defining term has been allocated.

If ILCS=YES is set, the register must contain the address of the value to be passed ("call by reference").

Description

PASS=STA

Parameters are passed via the STANDARD interface.

The PLIST operand specifies the actual parameters. In the LOCAL area of the calling procedure a parameter list containing the actual parameters is created.

Register 1 must not be used in the ADDR=reg operand.

PASS=OPT

Only allowed if ILCS=NO.
Parameters are passed via the OPTIMAL interface.

The PLIST operand specifies the actual parameters. Up to four parameters are passed to registers 1 through 4. When there are more than four parameters, the first three are passed to registers 2 through 4. For the rest, a parameter list is created whose address is passed to register 1.

Registers 1 to 4 must not be used in the ADDR=reg operand.

PLEND= Can only be specified in procedures in which ILCS=YES is set. Only to be used when specifying a PLIST.
Specifies whether the most significant bit is set for the last parameter in the parameter address list when parameters are passed dynamically.

YES The bit is set.

NO The bit is not set.

not specified:
If ILCS = NO, the bit is set for compatibility reasons;
if ILCS = YES, the bit is not set.

If ILCS=YES is set, register 0 is loaded with the number of parameters.

Register 1 is loaded with the address of a parameter list.

The most significant bit in the address of the last parameter in the parameter address list is set in accordance with the PLEND parameter.

When ILCS=YES, registers 0 and 1 are set to zero if no parameters are passed.

Format 3 of PASS is allowed in type M, E or I procedures only.

Programming notes

If a @PASS macro is called from within an ILCS procedure, the user is responsible for ensuring that contents of the parameter list are correct: ILCS always requires "call by reference". "Call by value" is also possible for non-ILCS procedures. No check can be made by the macros at assembly time, for example because of EQU symbols.

Example

The example shows the passing of three parameters via the STANDARD interface.

```
Name      | Operation | Operands
----------|-----------|-----------------------------------------------
PRO1      | @ENTR     | TYP=M,MAXPRM=3
          | @DATA     | CLASS=A,BASE=R6,DSECT=DUMMY
          | .         |
          | .         |
          | @PASS     | NAME=PRO2,PLIST=(FIELD1,(R7),FIELD4)
          | .         |
          | .         |
          | @EXIT     |
          | @END      |
DUMMY     | DSECT     |
FIELD1    | DS        | CL10
          | .         |
          | .         |
FIELD4    | DS        | CL25
LDUMMY    | EQU       | *-DUMMY
          | .         |
          | .         |
PRO2      | @ENTR     | TYP=I,LOCAL=IN,PLIST=(INPAR1,INPAR2,INPAR3)
          | .         |
          | .         |
          | @EXIT     |
          | @END      |
IN        | @PAR      | D=YES,LEND=YES,PLIST=(INPAR1,INPAR2,INPAR3)
          | .         |
          | .         |
```

>>>>> See also @ENTR and @PAR, format 2

@SETJV Set monitoring job variable

This macro call is only allowed in procedures with ILCS=YES.

Function

@SETJV (**SET J**ob **V**ariable) sets a value in the MONJV field of the PCD (see also section 9.6.7).

Format

Name	Operation	Operands
[name]	@SETJV	[MONJV={monjv / symb addr / (reg)}]

name Name
monjv Alphanumeric value
symb addr Symbolic address of a field
reg General-purpose register containing the address of the field.

Description

MONJV= Defines the new MONJV value

monjv Alphanumeric value, 4 characters in length, enclosed in single quotes.

symb addr1
Symbolic address of a field containing the string.

(reg1) Optionally, a register containing the address of the string.

not specified
Four blanks are entered in the PCD field MONJV.

Programming notes

1. If the string of the MONJV value is longer than 4 characters, the first 4 characters are used.
2. If the string of the MONJV value is shorter than 4 characters, the field is padded on the right with blanks.

@SETPM Set or reset program mask

This macro instruction is only allowed in procedures with ILCS=YES.

Function

@SETPM (**SET P**rogram **M**ask) sets the program mask or resets a changed program mask (see also section 9.6.6).

Format

Name	Operation	Operands
[name]	@SETPM	[PMASK={symb addr / (reg)}]

name Name
symb addr Symbolic address of a field
reg General-purpose register containing the program mask.

Description

PMASK= Defines the new program mask.

symb addr
Symbolic address of a 1-byte field containing the new program mask in hexadecimal form.

(reg) Optionally, a register containing the new program mask in the left byte.

not specified
The program mask is set to the ILCS default value.

Programming notes

If the program mask has been changed, it must be reset before an @EXIT, and whenever an ILCS procedure is called in accordance with ILCS conventions. The program mask must have the ILCS default value.

@STXDI Disable STXIT handling routine

This macro instruction is only allowed in procedures with ILCS=YES.

Function

@STXDI (**STX**IT **di**sable) deactivates an STXIT handling routine
(see also section 9.6.5, "STXIT handling").

Format

Name	Operation	Operands
[name]	@STXDI	STXID={symb addr / (reg)}

name Name
symb addr Sumbolic address of a field
reg General-purpose register containing the address of the ID code

Description

STXID ID code of the STXIT routine
The ID code is supplied by the @STXEN macro.

symb addr
Symbolic address of a 4-byte field aligned on a fullword boundary and containing the ID code.

(reg) Optionally, a register containing the address of the ID code.

The inline code of the macro copies the specifications from the macro instruction into an ILCS-conforming parameter block; these specifications are used to activate the corresponding entry in the standard STXIT handler for disabling the STXIT handling routine.

When the standard STXIT handler returns control to the calling procedure, register 15 contains a return code which indicates whether or not the function has been executed, and which errors, if any, occurred.

@STXEN Enable STXIT handling routine

This macro call is only allowed in procedures with ILCS=YES.

Function

@STXEN (**STX**IT **en**able) activates an STXIT handling routine (see also section 9.6.5, "STXIT handling").

Format

Name	Operation	Operands
[name]	@STXEN	STXNAME={name1 | symb addr1 | (reg1)} STXID={symb addr2 | (reg2)} STXADR={symb addr3 | (reg3)} [STXSVC={symb addr4 | (reg4)}]

name Name
name1 Alphabetic character string

symb addr1...symb addr4
Symbolic addresses of fields

reg1...reg4
General-purpose registers containing the address of a field.

Description

STXNAME= Name of the STXIT event

name1 'String' with up to 7 characters.

The following values can be specified for the corresponding interrupt classes:

```
'PROCHK ' :  "Program check"
'TIMER  ' :  "CPU interval timer"
'RUNOUT ' :  "End of program runtime"
'ERROR  ' :  "Unrecoverable program error"
'ABEND  ' :  Class "ABEND"
'ESCPBRK' :  Class "ESCPBRK"
'TERM   ' :  "Program termination"
'RTIMER ' :  "Real time interval timer"
'INTR   ' :  "Message to program"
'MSG    ' :  "Message to program"
'HWERROR' :  "Hardware error"
'SVC    ' :  "SVC interrupt"
```

symb addr1

Symbolic address of a 7-byte field containing one of the above strings.

(reg1)

Optionally, a register containing the address of the string.

STXID= ID code returned

symb addr2

Symbolic address of a 4-byte field aligned on a fullword boundary.

(reg2)

Optionally, a register containing the address of the return field.

STXADR= Start address of the STXIT routine

symb addr3

Symbolic address of a 4-byte field, aligned on a fullword boundary, containing the start address.

(reg3)

Optionally, a register containing the start address.

STXSVC= List of SVC numbers
This parameter must be specified if 'SVC' was specified as the event name in the first parameter.

symb addr4
Symbolic address of a list aligned on a halfword boundary, containing the number of entries and SVC numbers.

(reg4)
Optionally, a register containing the SVC list address.

The inline code of the macro copies the specifications from the macro instruction into an ILCS-conforming parameter block; these specifications are used to activate the corresponding entry in the standard STXIT handler for enabling the STXIT handling routine.

When the standard STXIT handler returns control to the calling procedure, register 15 contains a return code which indicates whether or not the function has been executed, and which errors, if any, occurred.

Programming notes

The 'HWERROR' and 'MSG' are not supported on releases earlier than BS2000 V11.0 (and they also require the correct version of ILCS).
The macro already maps 'MSG' onto 'INTR'.

@STXIM Define interrupt message layout

This macro instruction is only allowed in procedures with ILCS=YES.

Function

@STXIM (**STX**IT **I**nterrupt **M**essage) defines the layout of parameters that are passed to the user routine by the standard STXIT handler (SSH).

Format

Name	Operation	Operands
[name]	@STXIM	[pre]=string

name Name
pre Prefix
string Alphanumeric characters

Description

pre = Prefix for the symbolic names of the individual fields

string No more than 4 characters
pre must conform to the assembler syntax for names.

The prefix is used to define the following DS statements for the individual fields:

```
preSTIW  DS  F     STXIT interrupt weight
preSTMG  DS  CL64  Interrupt message
```

These parameters are passed to the user routine by the standard STXIT handler if an STXIT event occurs.

Programming notes

A prefix with more than 4 characters is truncated to 4 characters.

@THEN YES sub-block

Function

@THEN forms the heading of the YES sub-block in a decision.

Format

Name	Operation	Operands
[name]	@THEN	

Description

The sub-block in an @IF branch which begins with @THEN is executed **only** if the condition set with @IF holds true.

>>>>> See also @IF, @ELSE and @BEND

@THRU Iterative loop

Function

@THRU forms the heading of a loop construction. The number of executions is determined by specifying the initial and end values of a control variable.

Format

Name	Operation	Operands
[name]	@THRU	(reg1),{(reg2) / name2 / literal2}[,{(reg3) / name3 / literal3}]

name Name

reg1, reg2, reg3
: General-purpose registers; positive absolute expressions, either
  - decimal self-defining terms or
  - predefined names of registers or
  - names to which an appropriate self-defining term has been allocated

name2, name3
: Names of constants or memory areas aligned on a halfword boundary and which must always be one halfword long

literal2, literal3
: Literals aligned on a halfword boundary and which must always be one halfword long.

Description

reg1 contains the initial value for the control variable

reg2 or name2 or literal2
: specifies the end value for the control variable

reg3 or name3 or literal3
: specifies the increment.
  Default for the increment is 1.

The increment is added to the control variable after execution of the loop sub-block. If the sum of control variable and increment is greater than the end value, the structure block is quit.

Programming notes

1. reg1 must be loaded with the initial value before calling the @THRU macro.
2. If no increment is specified, register 0 must not be used for the initial value.
3. A @THRU macro with initial value ≤ end value and increment ≤ 0 results in a continuous loop.

Example

```
Name        Operation   Operands
            .
            .
            LH          R6,BEGL
LOOP        @THRU       (R6),ENDL
            @DO
            .
            .
            @BEND
            .
            .
EGL         DC          H'1'
ENDL        DC          H'10'
```

In this example, the initial value of the control variable is 1, the end value 10, and the increment also 1 (default). The loop sub-block is therefore executed 10 times.

>>>>> See also @DO and @BEND

@TOR Logical 'OR with priority'

Function

In compound conditions, @TOR implements a logical OR operation, which has a higher linkage priority than @AND.

Format

Name	Operation	Operands
[name]	@TOR	cond_sym

name Name
cond_sym Predefined or user-own condition symbol (see sections 9.2.4 and 9.2.5).

Programming notes

The call of the @TOR macro must be followed by a condition code setting machine instruction (see "Assembler Instructions" manual [3]).

Example

```
Name        Operation   Operands
LOOP        @CYCL
            .
            .
            @WHEN       EQ
            CR          R1,R2
            @AND        EQ
            CR          R7,R8
            @TOR        EQ
            CLI         FIELD,C'3'
            @BREA
            .
            .
            @BEND
```

The loop is quit, if either
- R1 = R2 and R7 = R8 or
- R1 = R2 and FIELD = 3.

>>>>> See also @AND and @OR

@WHEN Loop termination condition

Function

In a loop with unrestricted terminal condition or in a count loop with unrestricted terminal condition, @WHEN specifies the condition which results in termination of the loop.

Format

Name	Operation	Operands
[name]	@WHEN	cond_sym

name Name
cond_sym Predefined or user-own condition symbol (see sections 9.2.4 and 9.2.5).

Description

cond_sym specifies the condition (see section 9.2.2) which is to terminate the loop. If the condition holds true, the loop is exited with the @BREA macro.

Example See @BREA

>>>>> See also @BREA, @CYCL and @BEND

@WHIL Loop with pre-check

Function

@WHIL forms the heading of a loop construction in which the number of executions is determined by whether or not any condition prevails.

Format

Name	Operation	Operands
[name]	@WHIL[E]	cond_sym

name Name
cond_sym Predefined or user-own condition symbol (see sections 9.2.4 and 9.2.5).

Description

cond_sym specifies the condition (see sections 9.2.4 and 9.2.5) that initiates execution of the loop sub-block. When the @WHIL macro is called, the condition is checked and, if it holds true, the loop sub-block is executed.

Example

Name	Operation	Operands
LOOP	@WHIL	EQ
	CLI	END,C’N’
	@DO	
	.	
	.	
	@BEND	

The loop sub-block is executed provided END is equal to N.

>>>>> See also @DO and @BEND

11 Appendix

11.1 Summary of DC constants

Type	Specified by	Alignment	Implied Length (byte)	Length Modifier	Exponent Modifier	Scaling Factor	Padding, Truncation
A	absolute or relocatable expression	word	4	1 to 4			left
B	binary digits	byte	as required	1 to 256			left
C	characters	byte	as required	1 to 256			right
D	decimal digits	double word	8	1 to 8	-85 to +75	0 to 14	Padded right Truncation not applicable
E	decimal digits	word	4	1 to 8	-85 to +75	0 to 14	Padded right Truncation not applicable
F	decimal digits	word	4	1 to 8	-85 to +75	-187 to +346	left
H	decimal digits	halfword	2	1 to 8	-85 to +75	-187 to +346	left
L	decimal digits	double word	16	1 to 16	-85 to +75	0 to 28	Padded right Truncation not applicable
P	decimal digits	byte	as required	1 to 16			left
Q	name of a dummy register	word	4	1 to 4		left	
S	symbol or non-symbol	halfword	2	2			
V	name	word	4	3 or 4			left
X	hexadecimal digits	byte	as required	1 to 256			left
Y	absolute or relocatable expression	halfword	2	1 or 2		left	
Z	decimal digits	byte	as required	1 to 16			left

11.2 Format of the machine instructions

The instruction list below contains the instructions of the BS2000-NXS (SET1), BS2000-XS (SET3) and BS2000-ESA instruction sets (the Assembler instructions are described in the "Assembler Instructions" Language Reference Manual[ ]).
The BS2000-NXS instruction set supports systems with 24-bit addressing (NXS stands for Non-eXtended System).
The BS2000-XS instruction set supports XS systems with 31-bit addressing (XS stands for eXtended System).
The BS2000-ESA supports ESA systems, which allow for expansion of virtual address space (ESA stands for Enterprise Systems Architecture).
The BS2000-NXS instruction set is incorporated in the BS2000-XS instruction set, and both are incorporated the BS2000-ESA instruction set.
The instruction set to which each instruction belongs is indicated by the initial letter N, X or E in the NXS / XS / ESA column.
In the list below, the instructions marked N represent the basic instruction set, while those marked X or E belong to the corresponding extended instruction sets.

Mnemonic code	Instruction name	NXS XS ESA	Mach. code	Length	Operand format
A	Add	N	5A	4	R1,D2(X2,B2)
AD	Add normalized, long	N	6A	4	R1,D2(X2,B2)
ADR	Add normalized, long	N	2A	2	R1,R2
AE	Add normalized, short	N	7A	4	R1,D2(X2,B2)
AER	Add normalized, short	N	3A	2	R1,R2
AH	Add halfword	N	4A	4	R1,D2(X2,B2)
AL	Add logical	N	5E	4	R1,D2(X2,B2)
ALR	Add logical	N	1E	2	R1,R2
AP	Add decimal	N	FA	6	D1(L1,B1),D2(L2,B2)
AR	Add	N	1A	2	R1,R2
AU	Add unnormalized, short	N	7E	4	R1,D2(X2,B2)
AUR	Add unnormalized, short	N	3E	2	R1,R2
AW	Add unnormalized, long	N	6E	4	R1,D2(X2,B2)
AWR	Add unnormalized, long	N	2E	2	R1,R2
AXR	Add normalized with extended length	N	36	2	R1,R2
BAL	Branch and link	N	45	4	R1,D2(X2,B2)
BALR	Branch and link	N	05	2	R1,R2
BAS	Branch and link	N	4D	4	R1,D2(X2,B2)
BASR	Branch and link	N	0D	2	R1,R2
BASSM	Branch and save and set mode	X	0C	2	R1,R2
BC	Branch on condition	N	47	4	I,D2(X2,B2)
BCR	Branch on condition	N	07	2	I,R2
BCT	Branch on count	N	46	4	R1,D2(X2,B2)
BCTR	Branch on count	N	06	2	R1,R2
BSM	Branch and save	X	0B	2	R1,R2
BXH	Branch on index high	N	86	4	R1,R3,D2(B2)
BXLE	Branch on index low or equal	N	87	4	R1,R3,D2(B2)
C	Algebraic comparison	N	59	4	R1,D2(X2,B2)
* CCPU	Check CPU	N	AC	4	D1(B1),I2
CCW	Define channel command word	N		8	I1,I2,I3,I4
CCW0	Define channel command word (format 0)	X		8	I1,I2,I3,I4
CCW1	Define channel command word (format 1)	X		8	I1,I2,I3,I4
CD	Compare long	N	69	4	R1,D2(X2,B2)
CDR	Compare long	N	29	2	R1,R2
CDS	Compare double and swap	N	BB	4	R1,R3,D2(B2)
CE	Compare short	N	79	4	R1,D2(X2,B2)
CER	Compare short	N	39	2	R1,R2
CH	Compare halfword	N	49	4	R1,D2(C2,B2)
* CIOC	Check I/O controller	N	AD	4	D1(B1),I2
* CKC	Check channel	N	9F	4	D1(B1)
CL	Compare logical	N	55	4	R1,D2(X2,B2)
CLC	Compare logical	N	D5	6	D1(L,B1),D2(B2)
CLCL	Compare logical characters long	N	0F	2	R1,R2
CLI	Compare logical	N	95	4	D1(B1),I2
CLM	Compare logical chars. under mask	N	BD	4	R1,M3,D2(B2)

Mnemonic code	Instruction name	NXS XS ESA	Mach. code	Length	Operand format
CLM	Compare logical chars. under mask	N	BD	4	R1,M3,D2(B2)
CLR	Compare logical	N	15	2	R1,R2
CP	Compare decimal	N	F9	6	D1(L1,B1),D2(L2,B2)
CPYA	Copy Access Register	E	B24D	4	R1,R2
CR	Algebraic comparison	N	19	2	R1,R2
CS	Compare and swap	N	BA	4	R1,R3,D2(B2)
* CSCH	Clear subchannel	X	B230	4	No operand
CVB	Convert into binary form	N	4F	4	R1,D2(X2,B2)
CVD	Convert into decimal form	N	4E	4	R1,D2(X2,B2)
D	Divide	N	5D	4	R1,D2(X2,B2)
DD	Divide long	N	6D	4	R1,D2(X2,B2)
DDR	Divide long	N	2D	2	R1,R2
DE	Divide short	N	7D	4	R1,D2(X2,B2)
DER	Divide short	N	3D	2	R1,R2
* DIG	Diagnose	N	83	4	D1(B1)
DP	Divide decimal	N	FD	6	D1(L1,B1),D2(L2,B2)
DR	Divide	N	1D	2	R1,R2
DXR	Divide extended	X	B22D	4	R1,R2
EAR	Extract Access Register	E	B24F	4	R1,R2
ED	Edit	N	DE	6	D1(L,B1),D2(B2)
EDMK	Edit and mark	N	DF	6	D1(L,B1),D2(B2)
** EPAR	Extract primary ASN	X	B226	4	R1
** ESAR	Extract secondary AS	X	B227	4	R1
EX	Execute	N	44	4	R1,D2(X2,B2)
* FC	Execute special functions	N	9A	4	D1(B1),I2
* FCAL	Execute special functions	N	B7	4	D1(B1),I2
HDR	Halve long	N	24	2	R1,R2
* HDV	Halt device	N	9E	4	D1(B1)
HER	Halve short	N	34	2	R1,R2
* HSCH	Halt subchannel	X	B231	4	No operand
** IAC	Insert address space control	E	B224	4	R1
IC	Insert character	N	43	4	R1,D2(X2,B2)
ICM	Insert character with mask	N	BF	4	R1,M3,D2(B2)
* IDL	Idle	N	80	4	I2
** IPK	Insert PSW key	X	B208	4	No operand
IPM	Insert program mask	N	B222	4	R1
* ISK	Interrogate memory protect key	N	09	2	R1,R2
** IVSK	Insert virtual storage key	X	B223	4	R1,R2
L	Load	N	58	4	R1,D2(X2,B2)
LA	Load address	N	41	4	R1,D2(X2,B2)
LAE	Load Address Extended	E	51	4	R1,D2(X2,B2)
LAM	Load Access Multiple	E	9A	4	R1,R3,D2(B2)
LCDR	Load complement, long	N	23	2	R1,R2
LCER	Load complement, short	N	33	2	R1,R2
LCR	Load complement	N	13	2	R1,R2
LD	Load, long	N	68	4	R1,D2(X2,B2)
LDR	Load, long	N	28	2	R1,R2
LE	Load, short	N	78	4	R1,D2(X2,B2)
LER	Load, short	N	38	2	R1,R2
LH	Load halfword	N	48	4	R1,D2(X2,B2)
LM	Load multiple	N	98	4	R1,R3,D2(B2)

Mnemonic code	Instruction name	NXS XS ESA	Mach. code	Length	Operand format
		ESA			
LNR	Load negative	N	11	2	R1,R2
LPDR	Load positive, long	N	20	2	R1,R2
LPER	Load positive, short	N	30	2	R1,R2
LPR	Load positive	N	10	2	R1,R2
LR	Load	N	18	2	R1,R2
LRDR	Load rounded extended to long	N	25	2	R1,R2
LRER	Load rounded extended to short	N	35	2	R1,R2
* LSM	Load shadow memory	N	D9	6	D1(L,B1),D2(B2)
* LSP	Load scratch pad	N	D8	6	D1(L,B1),D2(B2)
LTDR	Load and test, long	N	22	2	R1,R2
LTER	Load and test, short	N	32	2	R1,R2
LTR	Load and test	N	12	2	R1,R2
M	Multiply	N	5C	4	R1,D2(X2,B2)
MD	Multiply, long	N	6C	4	R1,D2(X2,B2)
MDR	Multiply, long	N	2C	2	R1,R2
ME	Multiply, short	N	7C	4	R1,D2(X2,B2)
MER	Multiply, short	N	3C	2	R1,R2
MH	Multiply halfword	N	4C	4	R1,D2(X2,B2)
MP	Multiply decimal	N	FC	6	D1(L1,B1),D2(L2,B2)
MR	Multiply	N	1C	2	R1,R2
* MSCH	Modify subchannel	X	B232	4	D2(B2)
MVC	Move characters	N	D2	6	D1(L,B1),D2(B2)
MVCL	Move characters, long	N	0E	2	R1,R2
** MVCP	Move to primary	X	DA	6	D1(R1,B1),D2(B2),R3
** MVCS	Move to secondary	X	DB	6	D1(R1,B1),D2(B2),R3
MVI	Move immediate	N	92	4	D1(B1),I2
MVN	Move numerics	N	D1	6	D1(L,B1),D2(B2)
MVO	Move with offset	N	F1	6	D1(L1,B1),D2(L2,B2)
MVZ	Move zones	N	D3	6	D1(L,B1),D2(B2)
MXD	Multiply long to extended	N	67	4	R1,D2(X2,B2)
MXDR	Multiply long to extended	N	27	2	R1,R2
MXR	Multiply extended	N	26	2	R1,R2
N	AND	N	54	4	R1,D2(X2,B2)
NC	AND	N	D4	6	D1(L,B1),D2(B2)
NI	AND	N	94	4	D1(B1),I2
NR	AND	N	14	2	R1,R2
O	OR	N	56	4	R1,D2(X2,B2)
OC	OR	N	D6	6	D1(L,B1),D2(B2)
OI	OR	N	96	4	D1(B1),I2
OR	OR	N	16	2	R1,R2
PACK	Pack	N	F2	6	D1(L1,B1),D2(L2,B2)
** PC	Change function status	X	B218	4	D2(B2)
** PT	Program transfer	X	B228	4	R1,R2
* RCHP	Reset channel path	X	B23B	4	No operand
* RDD	Read direct	N	85	4	D1(B1),I2
* RSCH	Resume subchannel	X	B238	4	No operand
S	Subtract	N	5B	4	R1,D2(X2,B2)
SAC	Set address space control	E	B219	4	D2(B2)

Mnemonic code	Instruction name	NXS XS ESA	Mach. code	Length	Operand format
* SAL	Set address limit	X	B237	4	No operand
SAR	Set Access Register	E	B24E	4	R1,R2
* SCHM	Set channel monitor	X	B23C	4	No operand
SD	Subtract normalized, long	N	6B	4	R1,D2(X2,B2)
SDR	Subtract normalized, long	N	2B	2	R1,R2
* SDV	Start device	N	9C	4	D1(B1)
SE	Subtract normalized, short	N	7B	4	R1,D2(X2,B2)
SER	Subtract normalized, short	N	3B	2	R1,R2
SH	Subtract halfword	N	4B	4	R1,D2(X2,B2)
SL	Subtract logical	N	5F	4	R1,D2(X2,B2)
SLA	Shift left single	N	8B	4	R1,D2(B2)
SLDA	Shift left double	N	8F	4	R1,D2(B2)
SLDL	Shift left double logical	N	8D	4	R1,D2(B2)
SLL	Shift left single logical	N	89	4	R1,D2(B2)
SLR	Subtract without overflow	N	1F	2	R1,R2
SP	Subtract decimal	N	FB	6	D1(L1,B1),D2(L2,B2)
** SPKA	Set PSW key from address	X	B20A	4	D2(B2)
SPM	Set program mask	N	04	2	R1
SR	Subtract	N	1B	2	R1,R2
SRA	Shift right single	N	8A	4	R1,D2(B2)
SRDA	Shift right double	N	8E	4	R1,D2(B2)
SRDL	Shift right double logical	N	8C	4	R1,D2(B2)
SRL	Shift right single logical	N	88	4	R1,D2(B2)
SRP	Shift and round decimal	N	F0	6	D1(L1,B1),D2(B2),I3
* SSCH	Start subchannel	X	B233	4	D2(B2)
* SSK	Set memory protect key	N	08	2	R1,R2
* SSM	Store shadow memory	N	DA	6	D1(L,B1),D2(B2)
* SSP	Store scratch pad	N	D0	6	D1(L,B1),D2(B2)
ST	Store	N	50	4	R1,D2(X2,B2)
STAM	Store Access Multiple	E	9B	4	R1,R3,D2(B2)
STC	Store character	N	42	4	R1,D2(X2,B2)
STCK	Store clock	N	B2	4	D1(B1)
STCM	Store character with mask	N	BE	4	R1,M3,D2(B2)
* STCPS	Store channel path status	N	B23A	4	D2(B2)
* STCRW	Store channel report word	N	B239	4	D2(B2)
STD	Store long	N	60	4	R1,D2(X2,B2)
STE	Store short	N	70	4	R1,D2(X2,B2)
STH	Store halfword	N	40	4	R1,D2(X2,B2)
STM	Store multiple	N	90	4	R1,R3,D2(B2)
* STSCH	Store subchannel	X	B234	4	D2(B2)
SU	Subtract unnormalized, short	N	7F	4	R1,D2(X2,B2)
SUR	Subtract unnormalized, short	N	3F	2	R1,R2
SVC	Supervisor call	N	0A	2	I
SW	Subtract unnormalized, long	N	6F	4	R1,D2(X2,B2)
SWR	Subtract unnormalized, long	N	2F	2	R1,R2

Mnemonic code	Instruction name	NXS XS ESA	Mach. code	Length	Operand format
SXR	Subtract normalized extended	N	37	2	R1,R2
TAR	Test Access Register	E	B24C	4	R1,R2
* TDV	Test device	N	9D	4	D1(B1)
TM	Test under mask	N	91	4	D1(B1),I2
* TPI	Test pending interruption	X	B236	4	D2(B2)
TR	Translate	N	DC	6	D1(L,B1),D2(B2)
* TRACE	Trace	X	99	4	R1,R3,D2(B2)
TRT	Translate and test	N	DD	6	D1(L,B1),D2(B2)
TS	Test and set	N	93	4	D1(B1)
* TSCH	Test subchannel	X	B235	4	D2(B2)
UNPK	Unpack	N	F3	6	D1(L1,B1),D2(L2,B2)
* WRD	Write direct	N	84	4	D1(B1),I2
X	Exclusive-OR operation	N	57	4	R1,D2(X2,B2)
XC	Exclusive-OR operation	N	D7	6	D1(L,B1),D2(B2)
XI	Exclusive-OR operation	N	97	4	D1(B1),I2
XR	Exclusive-OR operation	N	17	2	R1,R2
ZAP	Zero and add	N	F8	6	D1(L1,B1),D2(L2,B2)

* Privileged instructions

** Semi-privileged instructions

11.3 Assembler restrictions

ASSEMBH cannot assemble a program of any size. There are restrictions with regard to program size, number of names, and so on, which must be adhered to in order to achieve a correct assembly. These ASSEMBH restrictions are detailed in this section. In addition to these theoretical maximum values, the size of programs which ASSEMBH can assemble is dependent on storage capacity and the complexity of the programs and the assembler.

- **Number of lines in the source program**

In the source program, every single instruction and every remark is regarded as a statement, and receives a statement number in the assembler listing. If an instruction is continued over several lines, the instruction still only has one statement number.

After resolution of the macros and insertion of COPY elements, a source program may contain a maximum of 2^{31}-1 statements.

- **Number of names and literals**

A source program may contain 2^{31}-1 names and literals, maximum, i.e. the symbol table may contain that number of entries at most.

If more names and literals have been used in a source program, ASSEMBH issues a message and assembly is abnormally terminated.

- **Length of names**

Names may have a maximum length of 64 characters. External names in modules in OM (object module) format, which are processed by the TSOSLNK linkage editor, are limited to eight characters.
External names in modules generated using the COMPILER-ACT(,MODULE-FORMAT=LLM) option (see ASSEMBH, User Guide [1]) are expanded to a length of 32 characters and are processed by the BINDER linkage editor.

- **Parenthesizing levels in arithmetic expressions**

The result of the expression

```
(number of elements) + (number of operators) + (parenthesizing levels) +
(management entries)
```

must not exceed the limits specified by the internal assembler tables.

- **Number of macro definitions**

From the libraries allocated for an assembly and the source text, a maximum 2^{31}-1 macro definitions can be used.

- **Nesting level of macros and COPY elements**

The maximum possible nesting level of macros is 255.

The nesting level for COPY elements is preset at 5. This can be increased to a maximum of 255 using the SDF option (see "ASSEMBH User Guide" [1]).

- **Length of macro instruction operands**

A macro instruction operand may be up to 1020 characters long.

In keyword operands, the number of characters after the equals sign is counted. If an operand contains a variable symbol, this length applies after insertion of the actual value. If the operand is an operand sublist, all commas and parentheses are counted as characters.

- **Variable symbols**

A maximum of 2^{15}-1 variable symbols may be used per macro definition.

- **Length of input records**

What the assembler is to interpret as the source text is determined when reading in from a file. This source text may be up to 255 characters in length.

The defaults for source text to be interpreted are columns 1, 71, 72, and 16. Using the SDF option FROM-COLUMN, TO-COLUMN, the setting for the begin column can be increased to 70, and that of the end column to a maximum of 255. With the ICTL instruction, the begin column can be set at 40, and the end column at a maximum of 80.

- **XREF listing**

There may be a maximum of 2^{31}-1 references in the XREF listing.

If this number is exceeded, an error message is issued and assembly is aborted.

- **ESID numbers**

ASSEMBH can issue a maximum of 2^{15}-1 ESID numbers.

An ESID number is issued for:
– each control section(Type SD)
– each common control section (Type CM)
– each dummy section(Type DS)
– each external dummy section (Types XD, XR)
– each XDSEC name (Types XD, XR)
– each EXTRN name (Type ER)
– each WXTRN name (Type WX)
– each V-type constant (Type VC)
– each dummy register (Type DX)

- **Assembler instructions/macro statements**

Number of LTORG instructions: any number
maximum duplication factor in DC and DS: 2^{24}-1
maximum length modifier in DC and DS: depends on constant type
maximum length specification in EQU: 2^{24}-1
maximum number of sequence symbols per macro: 2^{15}-1
maximum dimension in subscripted SET symbols: 2^{31}-1
maximum value range for arithmetic expressions: -2^{31} to 2^{31}-1
Number of continuation lines in an instr. statement: 9
Continuation lines in alternative statement format: any number

Other restrictions in structured programming

- **Predefined names**

All names beginning with "@" or "R@" are reserved for ASSEMBH, and may not be used elsewhere.

General-purpose registers can be accessed using R0 through R15, and floating-point registers with FA through FD.

Use of registers

Only registers 5 to 9 can be used without limitations. All others are reserved by the following register conventions.

Register	Utilization
1	Address of the parameter list for parameter passing
1 to 4	Parameter values or addresses for parameter passing in OPTIMAL form
10	Base address register for type M, E and I procedures
11	General-purpose register. For C programs: static areas pointer
12	Program Manager start address
13	Base address register for the STACK
14	Procedure return address
15	Procedure start address, Standard base address register for type B and L procedures

- **Names generated by ASSEMBH**

A maximum of 9000 internal names can be generated by ASSEMBH. If there is an overflow of the internal name count, ASSEMBH outputs an MNOTE with the termination weight.

- **Nesting of structure blocks**

The nesting of structure blocks is defined in a SETC symbol. Nesting is limited by the length of this symbol. If it is too long, an error message ensues.

- **Module size**

The size of an object module is limited by its addressability via a base address register.

11.4 Dummy registers: Examples

```
DUMMY REGISTER EXAMPLE 1/ PART 1                                                    14:27:36  1994-03-08   PAGE 0


LOCTN  OBJECT CODE    ADDR1    ADDR2     STMNT  M  SOURCE STATEMENT
                                             1             TITLE 'DUMMY REGISTER EXAMPLE 1/ PART 1'
                                             2     *
                                             3     * - IN THIS EXAMPLE THE LENGTH OF THE DUMMY REGISTER VECTOR (DRV) IS
                                             4     *   DEFINED AND STORED AT PROGRAMMING TIME.
                                             5     * - THE INDIVIDUAL DUMMY REGISTERS ARE ADDRESSED BY MEANS OF
                                             6     *   Q-TYPE CONSTANTS
                                             7     *
                                             8             PRINT NOGEN
000000                                       9     PSEUDO1  START
                                            10             EXTRN PSEUDO2
0000F4                                      11             ENTRY PSRVEKT
000000 05 A0                               12             BALR  10,0
000002                 00000002             13             USING *,10
                                            14     *
000002 41 10 A0F2      000000F4             15             LA    1,PSRVEKT
000006 58 20 A0E6      000000E8             16             L     2,QPSREG1               R2 <- DISPLACEMENT OF DUMMY REG1
00000A 1A 21                                17             AR    2,1                     RELATIVE TO START OF DRV
00000C D2 09 2000A142          00000144     18             MVC   0(10,2),=C'WUNDERLICH'  TO BECOME CONTENT OF PSREG1
000012 58 20 A0EE      000000F0             19             L     2,QPSREG3               R2 <- DISPLACEMENT OF DUMMY REG3
000016 1A 21                                20             AR    2,1                     RELATIVE TO START OF DRV
000018 D2 4F 2000A094          00000096     21             MVC   0(80,2),TEXT            TEXT TO BECOME CONTENT OF PSREG3
00001E 58 E0 A13E      00000140             22             L     14,=A(PSEUDO2)
000022 05 FE                                23             BALR  15,14
000024                                      24             WROUT NACHR1,FEHLER
                                            39   2               *,@DCEO      952    900503   53531004                    00066
                                            42   1               *,WROUT      004    890217   53121058
                                            43     *
000032                                      44             TERM
                                            47   2               *,VERSION 010                                            00001
000046                                      59     FEHLER   TERM  DUMP=Y
                                            62   2               *,VERSION 010                                            00001
                                            74     *
00005C                                      75     NACHR1   DS    0F
00005C 0038000040                           76             DC    X'0038000040'           X'38' = MESSAGE LENGTH
000061 5C5C5C5C5C5C5C5C                     77             DC    C'***************PROGRAMMENDE BEISPIEL 1****************'
000096 C5C9D540D4C5D5E2                     78     TEXT    DC    C'EIN MENSCH KANNS MANCHMAL NICHT VERSTEHN,'
0000BF E3D9C9C6C6E340C5                     79             DC    C'TRIFFT EIN WAS ER VORAUSGESEHN         '
                                            80     *
000000                                      81     PSREG1   DXD   CL10                   DEFINITIONS OF DUMMY REGISTERS 1
000000                                      82     PSREG2   DXD   CL60                                                  2
000000                                      83     PSREG3   DXD   20F                                                   3
                                            84     *
0000E8 00000000                             85     QPSREG1  DC    Q(PSREG1)              DEFINITIONS OF Q-TYPE CONSTANTS
0000EC 00000000                             86     QPSREG2  DC    Q(PSREG2)              FOR ADDRESSING THE INDIVIDUAL
0000F0 00000000                             87     QPSREG3  DC    Q(PSREG3)              DUMMY REGISTERS
                                            88     *
0000F4                                      89     PSRVEKT  DS    CL(L'PSREG1+L'PSREG2+L'PSREG3)    DUMMY REGISTER VECTOR
                                            90     *
                                            91             END
000140 00000000                             92                   =A(PSEUDO2)
000144 E6E4D5C4C5D9D3C9                     93                   =C'WUNDERLICH'


FLAGS IN 00000 STATEMENTS, 000 PRIVILEGED FLAGS, 000 MNOTES
HIGHEST ERROR-WEIGHT : NO ERRORS
THIS PROGRAM WAS ASSEMBLED BY ASSEMBH     V 1.2A00     ON 1994-03-08 AT 14:07:24
```

```
EXAMPLE 1/ PART 2                                                                    14:27:37  1994-03-08   PAGE 0

LOCTN  OBJECT CODE    ADDR1    ADDR2     STMNT  M  SOURCE STATEMENT
                                             1     *
                                             2               TITLE 'EXAMPLE 1/ PART 2'
                                             3     *
                                             4               PRINT NOGEN
000000                                       5     PSEUDO2   START
000000 05 B0                                 6               BALR  11,0
000002                 00000002              7               USING *,11
                                             8               EXTRN PSRVEKT
                                             9     *
000002 58 10 B016      00000018             10               L     1,=A(PSRVEKT)
000006 58 20 B012      00000014             11               L     2,QPSREG2               R2 <- DISPLACEMENT OF DUMMY REG2
00000A 1A 21                                 12               AR    2,1                     RELATIVE TO START OF DRV
00000C D2 13 2000B01A          0000001C      13               MVC   0(20,2),=C'(E.ROTH          )'
000012 07 FF                                 14               BR    15
                                             15    *
000000                                       16    PSREG2    DXD   CL22
000014 00000000                              17    QPSREG2   DC    Q(PSREG2)
                                             18    *
                                             19              END
000018 00000000                              20                    =A(PSRVEKT)
00001C 4DC54BD9D6E3C840                      21                    =C'(E.ROTH          )'


FLAGS IN 00000 STATEMENTS, 000 PRIVILEGED FLAGS, 000 MNOTES
HIGHEST ERROR-WEIGHT : NO ERRORS
THIS PROGRAM WAS ASSEMBLED BY ASSEMBH     V 1.2A00     ON 1994-03-08 AT 14:07:24
```

```
DUMMY REGISTER EXAMPLE 2/ PART 1                                                14:28:39  1994-03-08   PAGE 0

LOCTN  OBJECT CODE    ADDR1    ADDR2     STMNT  M  SOURCE STATEMENT
                                             1             TITLE 'DUMMY REGISTER EXAMPLE 2/ PART 1'
                                             2     *
                                             3     * - IN THIS EXAMPLE THE DUMMY REGISTER VECTOR (DRV) IS STORED
                                             4     *   AT PROGRAM RUNTIME BY REQUESTING MEMORY PAGES. THE PROGRAMMER
                                             5     *   NEED NOT CONCERN HIMSELF WITH THE LENGTH OF THE DRV, AS THIS IS
                                             6     *   ENTERED IN FIELD 'PRVLEN' (SEE INITIALISATION ROUTINE) BY THE
                                             7     *   LINKAGE EDITOR. EVALUATION IS PERFORMED AT PROGRAM RUNTIME.
                                             8     *
                                             9     * - ADDRESSING THE INDIVIDUAL DUMMY REGISTERS IS EFFECTED VIA
                                            10     *   A 'DUMMY REGISTER VECTOR BASE REGISTER'.
                                            11     *
                                            12             PRINT NOGEN
000000                                      13     PSEUDO1  START
                                            14             EXTRN PRVINIT
000000 05 A0                               15             BALR  10,0
000002                 00000002             16             USING *,10
000002                                      17             USING *PRV,8                 DEFINES THE BASE REGISTER FOR
                                            18     *                                    ACCESS TO DRV
000002 58 E0 A0E6      000000E8             19             L     14,=A(PRVINIT)
000006 05 CE                               20             BALR  12,14
000008 18 81                               21             LR    8,1                    R8 <- A(1ST ALLOCATED PAGE)
                                            22     *
00000A D2 23 8000A0EA 00000000 000000EC     23             MVC   PSREG1,=C'EIN MENSCH WIRD MUEDE SEINER FRAGEN:'
000010 D2 03 8000A07B 00000000 0000007D     24             MVC   PSREG2,TEXT2           MOVES DATA
000016 D2 20 8000A09F 00000000 000000A1     25             MVC   PSREG3,TEXT3           TO
00001C D2 07 8000A0C0 00000000 000000C2     26             MVC   PSREG4,TEXT4           DUMMY REGISTERS
000022                                      27             WROUT NACHR,FEHLER
                                            42   2              *,@DCEO      952    900503   53531004                    00066
                                            45   1              *,WROUT      004    890217   53121058
                                            46     *
000032                                      47     FEHLER   TERM
                                            50   2              *,VERSION 010                                            00001
                                            62     *
000048                                      63     NACHR    DS    0F
000048 0035000040                           64             DC    X'0035000040'
00004D 5C5C5C5C5C5C5C5C                     65             DC    C'*************PROGRAMMENDE BEISPIEL 2************'
00007D D5C9C540D2C1D5D5                     66     TEXT2    DC    C'NIE KANN DIE WELT IHM ANTWORT SAGEN.'
0000A1 C4D6C3C840C7C5D9                     67     TEXT3    DC    C'DOCH GERN GIBT AUSKUNFT ALLE WELT'
0000C2 C1E4C640C6D9C1C7                     68     TEXT4    DC    C'AUF FRAGEN, DIE ER NIE GESTELLT. '
                                            69     *
000000                                      70     PSREG1   DXD   CL36
000000                                      71     PSREG2   DXD   9F
000000                                      72     PSREG3   DXD   XL33
000000                                      73     PSREG4   DXD   4D
                                            74     *
000000                                      75             END   PSEUDO1
0000E8 00000000                             76                   =A(PRVINIT)
0000EC C5C9D540D4C5D5E2                     77                   =C'EIN MENSCH WIRD MUEDE SEINER FRAGEN:'


FLAGS IN 00000 STATEMENTS, 000 PRIVILEGED FLAGS, 000 MNOTES
HIGHEST ERROR-WEIGHT : NO ERRORS
THIS PROGRAM WAS ASSEMBLED BY ASSEMBH     V 1.2A00     ON 1994-03-08 AT 14:07:24
```

```
DUMMY REGISTER EXAMPLE 2/ INITIALIZATION                                                14:28:40  1994-03-08   PAGE 0

LOCTN  OBJECT CODE    ADDR1    ADDR2     STMNT  M  SOURCE STATEMENT
                                             1     *
                                             2               TITLE 'DUMMY REGISTER EXAMPLE 2/ INITIALIZATION'
                                             3               PRINT NOGEN
                                             4     *
000000                                       5     PRVINIT   START
000000 05 B0                                 6               BALR  11,0
000002                 00000002              7               USING *,11
                                             8     *
000002 17 22                                 9               XR    2,2                     DEFINES NUMBER OF 4K PAGES
000004 58 30 B036      00000038             10               L     3,PRVLEN                REQUIRED IN ACCORDANCE WITH
000008 5D 20 B03E      00000040             11               D     2,=F'4096'              LENGTH OF DRV AND ...
00000C 5A 30 B042      00000044             12               A     3,=F'1'
                                            13     *
000010                                      14               REQM  (3)                     ... ALLOCATION OF THIS
                                            17   2                 *,VERSION 500                                          00001
00001E 12 FF                                24               LTR   15,15
000020 07 8C                                25               BRZ   12
                                            26     *
000022                                      27               TERM  DUMP=Y
                                            30   2                 *,VERSION 010                                          00001
                                            42     *
000038                                      43               DS    0F
000038 00000000                             44     PRVLEN    CXD                           THIS IS WHERE THE LINKAGE EDITOR
                                            45     *                                     STORES THE LENGTH OF THE DRV
000000                                      46               END   PRVINIT
000040 00001000                             47                     =F'4096'
000044 00000001                             48                     =F'1'


FLAGS IN 00000 STATEMENTS, 000 PRIVILEGED FLAGS, 000 MNOTES
HIGHEST ERROR-WEIGHT : NOTE
THIS PROGRAM WAS ASSEMBLED BY ASSEMBH     V 1.2A00     ON 1994-03-08 AT 14:07:24
```

11.5 Parameter passing in structured programming: Example

```
ASSEMBH LISTING                                                                    14:29:44  1994-03-08   PAGE 0


LOCTN  OBJECT CODE    ADDR1    ADDR2     STMNT  M  SOURCE STATEMENT
000000                                       1     DEMOPARA START
                                             2           PRINT NOGEN
                                             3     * MATRIX PROCESSING OF WORD MATRIX STORED LINE-BY-LINE
000000                                       4     DEMOPARA @ENTR TYP=M,MAXPRM=2,                                  *
                                             4                  FUNCT='DEMO PARAMETER TRANSFER/ACCEPTANCE OPTIONS'
                                            88   3             *,VERSION 010                                    00001
                                           109     * CALCULATE MATRIX DIMENSION AND REQUEST STORAGE SPACE
000030 48 B0 A13E     00000140             110           LH    R11,NUMLINES
000034 4C B0 A140     00000142             111           MH    R11,NUMCOLS
000038 40 B0 A142     00000144             112           STH   R11,DIMMATR
00003C 1A BB                               113           AR    R11,R11
00003E 1A BB                               114           AR    R11,R11
000040                                     115           @DATA CLASS=A,BASE=R11,LENGTH=(R11)
                                           126     *
000050 41 60 A144     00000146             127           LA    ARLINE,HLINE
000054 41 70 A146     00000148             128           LA    ARCOLUMN,HCOLUMN
                                           129     * PROCESS MATRIX ELEMENT-BY-ELEMENT
000058 41 80 0001                          130           LA    RLINE,1
                                           131           @THRU (RLINE),NUMLINES
00005C                                     135           @DO
000064 40 80 A144     00000146             139           STH   RLINE,HLINE
000068 41 90 0001                          140           LA    RCOLUMN,1
                                           141           @THRU (RCOLUMN),NUMCOLS
00006C                                     145           @DO
000074 40 90 A146     00000148             149           STH   RCOLUMN,HCOLUMN
                                           150           PRINT GEN
                                           151           EJECT
```

```
ASSEMBH LISTING                                                    14:29:44  1994-03-08   PAGE 0

LOCTN  OBJECT CODE    ADDR1    ADDR2     STMNT  M  SOURCE STATEMENT
                                             152     * DYNAMIC PARAMETER TRANSFER (OPTIMAL)
                                             153     *
                                             154     * TRANSFER OF ADDRESS OF LINE AND COLUMN NUMBER AS REGISTER CONTENTS
                                             155     * REGISTERS R1, R2 REFER TO PARAMETER VALUES
                                             156               @PASS NAME=DO1,PASS=OPT,PLIST=((ARLINE),(ARCOLUMN))
                                             157   1           @@OUV ,,O,U,V,@PASS,(@,T,E,D,K,G,Y,A,X,Z,B)
                                             158   2           @@SYN K,@PASS,(@,T,E,D,K,G,Y,A,X,Z,B)
                                             159   1           @@PPP P,R@STACK,96,2,OPT,((ARLINE),(ARCOLUMN))
000078 18 16                                 160   2           LR    R1,ARLINE
00007A 18 27                                 161   2           LR    R2,ARCOLUMN
00007C 58 F0 A15A     0000015C               162   1           L     R@PASS,=A(DO1)
                                             163   1         ##BALR  R@EXIT,R@PASS
000080 05 EF                                 164   2           BALR  R@EXIT,R@PASS
                                             165     * TRANSFER OF ADDRESS OF LINE AND COLUMN NUMBER AS REGISTER CONTENTS
                                             166     * REGISTERS R1, R2 REFER TO PARAMETER VALUES
                                             167               @PASS NAME=DO2,PASS=OPT,PLIST=(HLINE,HCOLUMN)
                                             168   1           @@OUV ,,O,U,V,@PASS,(@,T,E,D,K,G,Y,A,X,Z,B)
                                             169   2           @@SYN K,@PASS,(@,T,E,D,K,G,Y,A,X,Z,B)
                                             170   1           @@PPP P,R@STACK,96,2,OPT,(HLINE,HCOLUMN)
000082 41 10 A144     00000146               171   2           LA    R1,HLINE
000086 41 20 A146     00000148               172   2           LA    R2,HCOLUMN
00008A 58 F0 A15E     00000160               173   1           L     R@PASS,=A(DO2)
                                             174   1         ##BALR  R@EXIT,R@PASS
00008E 05 EF                                 175   2           BALR  R@EXIT,R@PASS
                                             176     * TRANSFER OF ADDRESS OF LINE AND COLUMN NUMBER AS REGISTER CONTENTS
                                             177     * REGISTERS R1, R2 REFER TO PARAMETER VALUES
                                             178               @PASS NAME=DO3,PASS=OPT,PLIST=(HLINE,HCOLUMN)
                                             179   1           @@OUV ,,O,U,V,@PASS,(@,T,E,D,K,G,Y,A,X,Z,B)
                                             180   2           @@SYN K,@PASS,(@,T,E,D,K,G,Y,A,X,Z,B)
                                             181   1           @@PPP P,R@STACK,96,2,OPT,(HLINE,HCOLUMN)
000090 41 10 A144     00000146               182   2           LA    R1,HLINE
000094 41 20 A146     00000148               183   2           LA    R2,HCOLUMN
000098 58 F0 A162     00000164               184   1           L     R@PASS,=A(DO3)
                                             185   1         ##BALR  R@EXIT,R@PASS
00009C 05 EF                                 186   2           BALR  R@EXIT,R@PASS
                                             187     * TRANSFER OF LINE AND COLUMN NUMBER AS REGISTER CONTENTS
                                             188     * REGISTERS R1, R2 CONTAIN PARAMETER VALUES
                                             189               @PASS NAME=DO4,PASS=OPT,PLIST=((RLINE),(RCOLUMN))
                                             190   1           @@OUV ,,O,U,V,@PASS,(@,T,E,D,K,G,Y,A,X,Z,B)
                                             191   2           @@SYN K,@PASS,(@,T,E,D,K,G,Y,A,X,Z,B)
                                             192   1           @@PPP P,R@STACK,96,2,OPT,((RLINE),(RCOLUMN))
00009E 18 18                                 193   2           LR    R1,RLINE
0000A0 18 29                                 194   2           LR    R2,RCOLUMN
0000A2 58 F0 A166     00000168               195   1           L     R@PASS,=A(DO4)
                                             196   1         ##BALR  R@EXIT,R@PASS
0000A6 05 EF                                 197   2           BALR  R@EXIT,R@PASS
                                             198               EJECT
```

```
ASSEMBH LISTING                                                    14:29:44  1994-03-08   PAGE 0

LOCTN  OBJECT CODE    ADDR1    ADDR2     STMNT  M  SOURCE STATEMENT
                                           199     * DYNAMIC PARAMETER TRANSFER (STANDARD)
                                           200     *
                                           201     * TRANSFER OF ADDRESS OF DYNAMIC PARAMETER ADDRESS LIST
                                           202     * REGISTER 1 REFERS TO THE PARAMETER ADDRESS LIST
                                           203     * LIST ENTRIES REFER TO PARAMETER VALUES
                                           204               @PASS NAME=DS1,PASS=STA,PLIST=((ARLINE),(ARCOLUMN))
                                           205   1           @@OUV ,,O,U,V,@PASS,(@,T,E,D,K,G,Y,A,X,Z,B)
                                           206   2           @@SYN K,@PASS,(@,T,E,D,K,G,Y,A,X,Z,B)
                                           207   1           @@PPP P,R@STACK,96,2,STA,((ARLINE),(ARCOLUMN))
0000A8 50 60 D060                          208   2           ST    ARLINE,96(0,R@STACK)
0000AC 50 70 D064                          209   2           ST    ARCOLUMN,100(0,R@STACK)
0000B0 41 10 D060                          210   2           LA    R@PAR,96(0,R@STACK)
0000B4 96 80 D064                          211   2           OI    100(R@STACK),X'80'
0000B8 58 F0 A16A     0000016C             212   1           L     R@PASS,=A(DS1)
                                           213   1         ##BALR  R@EXIT,R@PASS
0000BC 05 EF                               214   2           BALR  R@EXIT,R@PASS
                                           215     * TRANSFER OF ADDRESS OF DYNAMIC PARAMETER ADDRESS LIST
                                           216     * REGISTER 1 REFERS TO THE PARAMETER ADDRESS LIST
                                           217     * LIST ENTRIES REFER TO PARAMETER VALUES
                                           218               @PASS NAME=DS2,PASS=STA,PLIST=(HLINE,HCOLUMN)
                                           219   1           @@OUV ,,O,U,V,@PASS,(@,T,E,D,K,G,Y,A,X,Z,B)
                                           220   2           @@SYN K,@PASS,(@,T,E,D,K,G,Y,A,X,Z,B)
                                           221   1           @@PPP P,R@STACK,96,2,STA,(HLINE,HCOLUMN)
0000BE 41 E0 A144     00000146             222   2           LA    R@EXIT,HLINE
0000C2 50 E0 D060                          223   2           ST    R@EXIT,96(0,R@STACK)
0000C6 41 E0 A146     00000148             224   2           LA    R@EXIT,HCOLUMN
0000CA 50 E0 D064                          225   2           ST    R@EXIT,100(0,R@STACK)
0000CE 41 10 D060                          226   2           LA    R@PAR,96(0,R@STACK)
0000D2 96 80 D064                          227   2           OI    100(R@STACK),X'80'
0000D6 58 F0 A16E     00000170             228   1           L     R@PASS,=A(DS2)
                                           229   1         ##BALR  R@EXIT,R@PASS
0000DA 05 EF                               230   2           BALR  R@EXIT,R@PASS
                                           231     * TRANSFER OF ADDRESS OF DYNAMIC PARAMETER ADDRESS LIST
                                           232     * REGISTER 1 REFERS TO THE PARAMETER ADDRESS LIST
                                           233     * LIST ENTRIES REFER TO PARAMETER VALUES
                                           234               @PASS NAME=DS3,PASS=STA,PLIST=(HLINE,HCOLUMN)
                                           235   1           @@OUV ,,O,U,V,@PASS,(@,T,E,D,K,G,Y,A,X,Z,B)
                                           236   2           @@SYN K,@PASS,(@,T,E,D,K,G,Y,A,X,Z,B)
                                           237   1           @@PPP P,R@STACK,96,2,STA,(HLINE,HCOLUMN)
0000DC 41 E0 A144     00000146             238   2           LA    R@EXIT,HLINE
0000E0 50 E0 D060                          239   2           ST    R@EXIT,96(0,R@STACK)
0000E4 41 E0 A146     00000148             240   2           LA    R@EXIT,HCOLUMN
0000E8 50 E0 D064                          241   2           ST    R@EXIT,100(0,R@STACK)
0000EC 41 10 D060                          242   2           LA    R@PAR,96(0,R@STACK)
0000F0 96 80 D064                          243   2           OI    100(R@STACK),X'80'
0000F4 58 F0 A172     00000174             244   1           L     R@PASS,=A(DS3)
                                           245   1         ##BALR  R@EXIT,R@PASS
0000F8 05 EF                               246   2           BALR  R@EXIT,R@PASS
                                           247               EJECT
```

```
ASSEMBH LISTING                                                                14:29:44  1994-03-08   PAGE 0

LOCTN  OBJECT CODE    ADDR1    ADDR2     STMNT  M  SOURCE STATEMENT
                                           248     * STATIC PARAMETER TRANSFER
                                           249     *
                                           250     * TRANSFER OF ADDRESS OF STATIC PARAMETER ADDRESS LIST
                                           251     * REGISTER 1 REFERS TO THE PARAMETER ADDRESS LIST
                                           252     * LIST ENTRIES REFER TO PARAMETER VALUES
                                           253               @PASS NAME=S1,PAR=STAPLIS
                                           254   1           @@OUV ,,O,U,V,@PASS,(@,T,E,D,K,G,Y,A,X,Z,B)
                                           255   2           @@SYN K,@PASS,(@,T,E,D,K,G,Y,A,X,Z,B)
0000FA 41 10 A12E     00000130             256   1           LA    R@PAR,STAPLIS
0000FE 58 F0 A176     00000178             257   1           L     R@PASS,=A(S1)
                                           258   1         ##BALR  R@EXIT,R@PASS
000102 05 EF                               259   2           BALR  R@EXIT,R@PASS
                                           260     * TRANSFER OF ADDRESS OF STATIC PARAMETER ADDRESS LIST
                                           261     * REGISTER 1 REFERS TO THE PARAMETER ADDRESS LIST
                                           262     * LIST ENTRIES REFER TO PARAMETER VALUES
                                           263               @PASS NAME=S2,PAR=STAPLIS
                                           264   1           @@OUV ,,O,U,V,@PASS,(@,T,E,D,K,G,Y,A,X,Z,B)
                                           265   2           @@SYN K,@PASS,(@,T,E,D,K,G,Y,A,X,Z,B)
000104 41 10 A12E     00000130             266   1           LA    R@PAR,STAPLIS
000108 58 F0 A17A     0000017C             267   1           L     R@PASS,=A(S2)
                                           268   1         ##BALR  R@EXIT,R@PASS
00010C 05 EF                               269   2           BALR  R@EXIT,R@PASS
                                           270     * TRANSFER OF ADDRESS OF STATIC PARAMETER ADDRESS LIST
                                           271     * REGISTER 1 REFERS TO THE PARAMETER ADDRESS LIST
                                           272     * LIST ENTRIES REFER TO PARAMETER VALUES
                                           273               @PASS NAME=S3,PAR=STAPLIS
                                           274   1           @@OUV ,,O,U,V,@PASS,(@,T,E,D,K,G,Y,A,X,Z,B)
                                           275   2           @@SYN K,@PASS,(@,T,E,D,K,G,Y,A,X,Z,B)
00010E 41 10 A12E     00000130             276   1           LA    R@PAR,STAPLIS
000112 58 F0 A17E     00000180             277   1           L     R@PASS,=A(S3)
                                           278   1         ##BALR  R@EXIT,R@PASS
000116 05 EF                               279   2           BALR  R@EXIT,R@PASS
                                           280               PRINT NOGEN
000118                                     281               @BEND
000120                                     288               @BEND
000128                                     295               @EXIT
                                           302               PRINT GEN
                                           303     * STATIC PARAMETER ADDRESS LIST
                                           304     STAPLIS   @PAR  PLIST=(LNA,CNA),VLIST=(HLINE,HCOLUMN)
                                           305   1           @@SYN ,@PAR,,LE,1
000130                                     306   1 STAPLIS   DS    0F
000130 00000146                            307   1 LNA       DC    A(HLINE)
000134 80000148                            308   1 CNA       DC    A(HCOLUMN+X'80000000')
                                           309               EJECT
```

```
ASSEMBH LISTING                                                                    14:29:44  1994-03-08   PAGE 0

LOCTN  OBJECT CODE    ADDR1    ADDR2     STMNT  M  SOURCE STATEMENT
                                           310             PRINT NOGEN
                      00000138             311     VAR     EQU   *
000138 00000146                            312     ADRLINE DC    A(HLINE)
00013C 00000148                            313     ADRCOL  DC    A(HCOLUMN)
000140                                     314     NUMLINES DS   H
000142                                     315     NUMCOLS DS    H
000144                                     316     DIMMATR DS    H
000146                                     317     HLINE   DS    H
000148                                     318     HCOLUMN DS    H
                      00000006             319     ARLINE  EQU   R6
                      00000007             320     ARCOLUMN EQU  R7
                      00000008             321     RLINE   EQU   R8
                      00000009             322     RCOLUMN EQU   R9
000150                                     323             @END
                      00000009             342     L       EQU   R9
                      00000008             343     C       EQU   R8
                      00000007             344     P       EQU   R7
                                           345             PRINT GEN
                                           346     *
                                           347     * PARAMETER TRANSFER VIA @PASS OF CALLING PROCEDURE
                                           348     *
                                           349     * PARAMETER ACCEPTANCE VIA @ENTR OF CALLED PROCEDURE
                                           350             EJECT
```

```
ASSEMBH LISTING                                                                    14:29:44  1994-03-08   PAGE 0

LOCTN  OBJECT CODE    ADDR1    ADDR2     STMNT  M  SOURCE STATEMENT
                                           351     * REGISTERS R1, R2 CONTAIN PARAMETER ADDRESSES
                                           352     DO1     @ENTR TYP=I,TITLE=NO
                                           353   1         @@SYN ,@ENTR,,EQ,0
000188                                     354   1 DO1     DS    0D
000188                00000000             355   1         USING @SAV,R@STACK
000188 90 EC D00C     0000000C             356   1         STM   R14,R12,@SAVR14
00018C 18 AF                               357   1         LR    R@BASE,R@PASS
00018E                00000188             358   1         USING DO1,R@BASE
                                           359   1         @PASS EXTNAME=$NUCENTR,CNOP=(0,4),DC=(A(96),CL8'DO1')
00018E 58 F0 A038     000001C0             360   2         L     R@PASS,=V($NUCENTR)
000192                                     361   2         CNOP  2,4
                                           362   2       ##BALR  R@EXIT,R@PASS
000192 05 EF                               363   3         BALR  R@EXIT,R@PASS
000194 00000060                            364   2         DC    A(96)
000198 C4D6F14040404040                    365   2         DC    CL8'DO1'
                                           366             PRINT NOGEN
0001A0                                     367             @DATA CLASS=S,BASE=R12,INIT=VAR
                                           372     * CALCULATE MATRIX POSITION
0001A4 48 70 1000                          373             LH    P,0(0,R1)     VALUE OF FIRST PARAMETER
0001A8 06 70                               374             BCTR  P,0
0001AA 4C 70 C00A     00000142             375             MH    P,NUMCOLS
0001AE 4A 70 2000                          376             AH    P,0(0,R2)     VALUE OF SECOND PARAMETER
0001B2 06 70                               377             BCTR  P,0
0001B4 1A 77                               378             AR    P,P
0001B6 1A 77                               379             AR    P,P
                                           380     * PERFORM FUNCTION
0001B8                                     381             @EXIT
0001C0                                     389             @END
                                           397             PRINT GEN
                                           398             EJECT
```

```
ASSEMBH LISTING                                                                14:29:44  1994-03-08   PAGE 0

LOCTN  OBJECT CODE    ADDR1    ADDR2     STMNT  M  SOURCE STATEMENT
                                             399     * ACCEPTANCE OF PARAMETER ADDRESSES FROM REGISTERS R1, R2
                                             400     *                                    INTO ADDRESS LIST
                                             401     DO2      @ENTR TYP=I,TITLE=NO,LOCAL=PARLI1,PASS=OPT,PLIST=(ADRLN,ADRCN)
                                             402   1          @@SYN ,@ENTR,,EQ,0
0001D0                                       403   1 DO2      DS    0D
0001D0                 00000000              404   1          USING @SAV,R@STACK
0001D0 90 EC D00C      0000000C              405   1          STM   R14,R12,@SAVR14
0001D4 18 AF                                 406   1          LR    R@BASE,R@PASS
0001D6                 000001D0              407   1          USING DO2,R@BASE
                                             408   1          @PASS EXTNAME=$NUCENTR,CNOP=(0,4),DC=(A(LPARLI1),CL8'DO2')
0001D6 58 F0 A048      00000218              409   2          L     R@PASS,=V($NUCENTR)
0001DA                                       410   2          CNOP  2,4
                                             411   2        ##BALR  R@EXIT,R@PASS
0001DA 05 EF                                 412   3          BALR  R@EXIT,R@PASS
0001DC 00000068                              413   2          DC    A(LPARLI1)
0001E0 C4D6F24040404040                      414   2          DC    CL8'DO2'
                                             415   1          @DATA BASE=R@STACK,DSECT=PARLI1
0001E8                 00000000              416   2          USING PARLI1,R@STACK
                                             417   1          @@PPP E,R@PAR,0,0,OPT,(ADRLN,ADRCN)
0001E8 50 10 D064      00000064              418   2          ST    R1,ADRLN
0001EC 50 20 D060      00000060              419   2          ST    R2,ADRCN
                                             420              PRINT NOGEN
0001F0                                       421              @DATA CLASS=S,BASE=R12,INIT=VAR
                                             426     * CALCULATE MATRIX POSITION
0001F4 58 20 D064      00000064              427              L     R2,ADRLN   ADDRESS OF FIRST PARAMETER
0001F8 48 70 2000                            428              LH    P,0(0,R2)    VALUE OF FIRST PARAMETER
0001FC 06 70                                 429              BCTR  P,0
0001FE 4C 70 C00A      00000142              430              MH    P,NUMCOLS
000202 58 20 D060      00000060              431              L     R2,ADRCN   ADDRESS OF SECOND PARAMETER
000206 4A 70 2000                            432              AH    P,0(0,R2)    VALUE OF SECOND PARAMETER
00020A 06 70                                 433              BCTR  P,0
00020C 1A 77                                 434              AR    P,P
00020E 1A 77                                 435              AR    P,P
                                             436     * PERFORM FUNCTION
000210                                       437              @EXIT
000218                                       445              @END
                                             453              PRINT GEN
                                             454     PARLI1   @PAR  D=YES,LEND=YES,PLIST=(ADRCN,ADRLN)
                                             455   1          @@SYN ,@PAR,,LE,1
000000                                       456   1 PARLI1   DSECT
000000                 00000060              457   1          ORG   *+96
000060                                       458   1 ADRCN    DS    A
000064                                       459   1 ADRLN    DS    A
                       00000068              460   1 LPARLI1  EQU   *-PARLI1
000228                                       461   1 DEMOPARA CSECT
                                             462              EJECT
```

```
ASSEMBH LISTING                                                    14:29:44  1994-03-08   PAGE 0

LOCTN  OBJECT CODE    ADDR1    ADDR2     STMNT  M  SOURCE STATEMENT
                                           463     * ACCEPTANCE OF PARAMETER ADDRESSES FROM REGISTERS R1, R2
                                           464     *                                    INTO OTHER REGISTERS
                                           465     DO3      @ENTR TYP=I,TITLE=NO,PASS=OPT,PLIST=((L),(C))
                                           466   1          @@SYN ,@ENTR,,EQ,0
000228                                     467   1 DO3      DS    0D
000228                 00000000            468   1          USING @SAV,R@STACK
000228 90 EC D00C      0000000C            469   1          STM   R14,R12,@SAVR14
00022C 18 AF                               470   1          LR    R@BASE,R@PASS
00022E                 00000228            471   1          USING DO3,R@BASE
                                           472   1          @PASS EXTNAME=$NUCENTR,CNOP=(0,4),DC=(A(96),CL8'DO3')
00022E 58 F0 A040      00000268            473   2          L     R@PASS,=V($NUCENTR)
000232                                     474   2          CNOP  2,4
                                           475   2        ##BALR  R@EXIT,R@PASS
000232 05 EF                               476   3          BALR  R@EXIT,R@PASS
000234 00000060                            477   2          DC    A(96)
000238 C4D6F34040404040                    478   2          DC    CL8'DO3'
                                           479   1          @@PPP E,R@PAR,0,0,OPT,((L),(C))
000240 18 91                               480   2          LR    L,R1
000242 18 82                               481   2          LR    C,R2
                                           482              PRINT NOGEN
000244                                     483              @DATA CLASS=S,BASE=R12,INIT=VAR
                                           488     * CALCULATE MATRIX POSITION
000248 48 70 9000                          489              LH    P,0(0,L)      VALUE OF FIRST PARAMETER
00024C 06 70                               490              BCTR  P,0
00024E 4C 70 C00A      00000142            491              MH    P,NUMCOLS
000252 4A 70 8000                          492              AH    P,0(0,C)      VALUE OF SECOND PARAMETER
000256 06 70                               493              BCTR  P,0
000258 1A 77                               494              AR    P,P
00025A 1A 77                               495              AR    P,P
                                           496     * PERFORM FUNCTION
00025C                                     497              @EXIT
000268                                     505              @END
                                           513              PRINT GEN
                                           514              EJECT
```

```
ASSEMBH LISTING                                                    14:29:44  1994-03-08   PAGE 0

LOCTN  OBJECT CODE    ADDR1    ADDR2     STMNT  M  SOURCE STATEMENT
                                           515     * ACCEPTANCE OF PARAMETER VALUES FROM REGISTERS R1, R2
                                           516     *                                INTO OTHER REGISTERS
                                           517     DO4      @ENTR TYP=I,TITLE=NO,PASS=OPT,PLIST=((L),(C))
                                           518   1          @@SYN ,@ENTR,,EQ,0
000278                                     519   1 DO4      DS    0D
000278                 00000000            520   1          USING @SAV,R@STACK
000278 90 EC D00C      0000000C            521   1          STM   R14,R12,@SAVR14
00027C 18 AF                               522   1          LR    R@BASE,R@PASS
00027E                 00000278            523   1          USING DO4,R@BASE
                                           524   1          @PASS EXTNAME=$NUCENTR,CNOP=(0,4),DC=(A(96),CL8'DO4')
00027E 58 F0 A038      000002B0            525   2          L     R@PASS,=V($NUCENTR)
000282                                     526   2          CNOP  2,4
                                           527   2        ##BALR  R@EXIT,R@PASS
000282 05 EF                               528   3          BALR  R@EXIT,R@PASS
000284 00000060                            529   2          DC    A(96)
000288 C4D6F44040404040                    530   2          DC    CL8'DO4'
                                           531   1          @@PPP E,R@PAR,0,0,OPT,((L),(C))
000290 18 91                               532   2          LR    L,R1
000292 18 82                               533   2          LR    C,R2
                                           534              PRINT NOGEN
000294                                     535              @DATA CLASS=S,BASE=R12,INIT=VAR
                                           540     * CALCULATE MATRIX POSITION
000298 18 79                               541              LR    P,L           VALUE OF FIRST PARAMETER
00029A 06 70                               542              BCTR  P,0
00029C 4C 70 C00A      00000142            543              MH    P,NUMCOLS
0002A0 1A 78                               544              AR    P,C           VALUE OF SECOND PARAMETER
0002A2 06 70                               545              BCTR  P,0
0002A4 1A 77                               546              AR    P,P
0002A6 1A 77                               547              AR    P,P
                                           548     * PERFORM FUNCTION
0002A8                                     549              @EXIT
0002B0                                     557              @END
                                           565              PRINT GEN
                                           566              EJECT
```

```
ASSEMBH LISTING                                                                14:29:44  1994-03-08   PAGE 0

LOCTN  OBJECT CODE    ADDR1    ADDR2     STMNT  M  SOURCE STATEMENT
                                             567     * REGISTER R1 CONTAINS ADDRESS OF PARAMETER ADDRESS LIST
                                             568     DS1      @ENTR TYP=I,TITLE=NO
                                             569   1          @@SYN ,@ENTR,,EQ,0
0002C0                                       570   1 DS1      DS    0D
0002C0                 00000000              571   1          USING @SAV,R@STACK
0002C0 90 EC D00C      0000000C              572   1          STM   R14,R12,@SAVR14
0002C4 18 AF                                 573   1          LR    R@BASE,R@PASS
0002C6                 000002C0              574   1          USING DS1,R@BASE
                                             575   1          @PASS EXTNAME=$NUCENTR,CNOP=(0,4),DC=(A(96),CL8'DS1')
0002C6 58 F0 A040      00000300              576   2          L     R@PASS,=V($NUCENTR)
0002CA                                       577   2          CNOP  2,4
                                             578   2        ##BALR  R@EXIT,R@PASS
0002CA 05 EF                                 579   3          BALR  R@EXIT,R@PASS
0002CC 00000060                              580   2          DC    A(96)
0002D0 C4E2F14040404040                      581   2          DC    CL8'DS1'
                                             582              PRINT NOGEN
0002D8                                       583              @DATA CLASS=S,BASE=R12,INIT=VAR
                                             588     * CALCULATE MATRIX POSITION
0002DC 58 20 1000                            589              L     R2,0(0,R1) ADDRESS OF FIRST PARAMETER
0002E0 48 70 2000                            590              LH    P,0(0,R2)    VALUE OF FIRST PARAMETER
0002E4 06 70                                 591              BCTR  P,0
0002E6 4C 70 C00A      00000142              592              MH    P,NUMCOLS
0002EA 58 20 1004                            593              L     R2,4(0,R1) ADDRESS OF SECOND PARAMETER
0002EE 4A 70 2000                            594              AH    P,0(0,R2)    VALUE OF SECOND PARAMETER
0002F2 06 70                                 595              BCTR  P,0
0002F4 1A 77                                 596              AR    P,P
0002F6 1A 77                                 597              AR    P,P
                                             598     * PERFORM FUNCTION
0002F8                                       599              @EXIT
000300                                       607              @END
                                             615              PRINT GEN
                                             616              EJECT
```

```
ASSEMBH LISTING                                                                14:29:44  1994-03-08   PAGE 0

LOCTN  OBJECT CODE    ADDR1    ADDR2     STMNT  M  SOURCE STATEMENT
                                           617     * ACCEPTANCE OF ADDRESSES FROM PARAMETER ADDRESS LIST
                                           618     *                          INTO ANOTHER ADDRESS LIST
                                           619     DS2      @ENTR TYP=I,TITLE=NO,LOCAL=PARLI2,PASS=STA,PLIST=(LNADR,CNADR)
                                           620   1          @@SYN ,@ENTR,,EQ,0
000310                                     621   1 DS2      DS    0D
000310                 00000000            622   1          USING @SAV,R@STACK
000310 90 EC D00C      0000000C            623   1          STM   R14,R12,@SAVR14
000314 18 AF                               624   1          LR    R@BASE,R@PASS
000316                 00000310            625   1          USING DS2,R@BASE
                                           626   1          @PASS EXTNAME=$NUCENTR,CNOP=(0,4),DC=(A(LPARLI2),CL8'DS2')
000316 58 F0 A050      00000360            627   2          L     R@PASS,=V($NUCENTR)
00031A                                     628   2          CNOP  2,4
                                           629   2        ##BALR  R@EXIT,R@PASS
00031A 05 EF                               630   3          BALR  R@EXIT,R@PASS
00031C 00000068                            631   2          DC    A(LPARLI2)
000320 C4E2F24040404040                    632   2          DC    CL8'DS2'
                                           633   1          @DATA BASE=R@STACK,DSECT=PARLI2
000328                 00000000            634   2          USING PARLI2,R@STACK
                                           635   1          @@PPP E,R@PAR,0,0,STA,(LNADR,CNADR)
000328 D2 03 D0641000 00000064             636   2          MVC   LNADR(4),0(R@PAR)
00032E D2 03 D0601004 00000060             637   2          MVC   CNADR(4),4(R@PAR)
                                           638              PRINT NOGEN
000334                                     639              @DATA CLASS=S,BASE=R12,INIT=VAR
                                           644     * CALCULATE MATRIX POSITION
000338 58 20 D064      00000064            645              L     R2,LNADR   ADDRESS OF FIRST PARAMETER
00033C 48 70 2000                          646              LH    P,0(0,R2)    VALUE OF FIRST PARAMETER
000340 06 70                               647              BCTR  P,0
000342 4C 70 C00A      00000142            648              MH    P,NUMCOLS
000346 58 20 D060      00000060            649              L     R2,CNADR   ADDRESS OF SECOND PARAMETER
00034A 4A 70 2000                          650              AH    P,0(0,R2)    VALUE OF SECOND PARAMETER
00034E 06 70                               651              BCTR  P,0
000350 1A 77                               652              AR    P,P
000352 1A 77                               653              AR    P,P
                                           654     * PERFORM FUNCTION
000354                                     655              @EXIT
000360                                     663              @END
                                           671              PRINT GEN
                                           672     PARLI2   @PAR  D=YES,LEND=YES,PLIST=(CNADR,LNADR)
                                           673   1          @@SYN ,@PAR,,LE,1
000000                                     674   1 PARLI2   DSECT
000000                 00000060            675   1          ORG   *+96
000060                                     676   1 CNADR    DS    A
000064                                     677   1 LNADR    DS    A
                       00000068            678   1 LPARLI2  EQU   *-PARLI2
000370                                     679   1 DEMOPARA CSECT
                                           680              EJECT
```

```
ASSEMBH LISTING                                                    14:29:44  1994-03-08   PAGE 0

LOCTN  OBJECT CODE    ADDR1    ADDR2     STMNT  M  SOURCE STATEMENT
                                           681     * ACCEPTANCE OF ADDRESSES FROM PARAMETER ADDRESS LIST
                                           682     *                          INTO REGISTERS
                                           683     DS3      @ENTR TYP=I,TITLE=NO,PASS=STA,PLIST=((R2),(R1))
                                           684   1          @@SYN ,@ENTR,,EQ,0
000370                                     685   1 DS3      DS    0D
000370                 00000000            686   1          USING @SAV,R@STACK
000370 90 EC D00C      0000000C            687   1          STM   R14,R12,@SAVR14
000374 18 AF                               688   1          LR    R@BASE,R@PASS
000376                 00000370            689   1          USING DS3,R@BASE
                                           690   1          @PASS EXTNAME=$NUCENTR,CNOP=(0,4),DC=(A(96),CL8'DS3')
000376 58 F0 A040      000003B0            691   2          L     R@PASS,=V($NUCENTR)
00037A                                     692   2          CNOP  2,4
                                           693   2        ##BALR  R@EXIT,R@PASS
00037A 05 EF                               694   3          BALR  R@EXIT,R@PASS
00037C 00000060                            695   2          DC    A(96)
000380 C4E2F34040404040                    696   2          DC    CL8'DS3'
                                           697   1          @@PPP E,R@PAR,0,0,STA,((R2),(R1))
000388 58 20 1000                          698   2          L     R2,0(0,R@PAR)
00038C 58 10 1004                          699   2          L     R1,4(0,R@PAR)
                                           700              PRINT NOGEN
000390                                     701              @DATA CLASS=S,BASE=R12,INIT=VAR
                                           706     * CALCULATE MATRIX POSITION
000394 48 70 2000                          707              LH    P,0(0,R2)     VALUE OF FIRST PARAMETER
000398 06 70                               708              BCTR  P,0
00039A 4C 70 C00A      00000142            709              MH    P,NUMCOLS
00039E 4A 70 1000                          710              AH    P,0(0,R1)     VALUE OF SECOND PARAMETER
0003A2 06 70                               711              BCTR  P,0
0003A4 1A 77                               712              AR    P,P
0003A6 1A 77                               713              AR    P,P
                                           714     * PERFORM FUNCTION
0003A8                                     715              @EXIT
0003B0                                     723              @END
                                           731              PRINT GEN
                                           732              EJECT
```

```
ASSEMBH LISTING                                                                14:29:44  1994-03-08   PAGE 0

LOCTN  OBJECT CODE    ADDR1    ADDR2     STMNT  M  SOURCE STATEMENT
                                             733     * REGISTER R1 CONTAINS ADDRESS OF PARAMETER ADDRESS LIST
                                             734     S1       @ENTR TYP=I,TITLE=NO
                                             735   1          @@SYN ,@ENTR,,EQ,0
0003C0                                       736   1 S1       DS    0D
0003C0                 00000000              737   1          USING @SAV,R@STACK
0003C0 90 EC D00C      0000000C              738   1          STM   R14,R12,@SAVR14
0003C4 18 AF                                 739   1          LR    R@BASE,R@PASS
0003C6                 000003C0              740   1          USING S1,R@BASE
                                             741   1          @PASS EXTNAME=$NUCENTR,CNOP=(0,4),DC=(A(96),CL8'S1')
0003C6 58 F0 A040      00000400              742   2          L     R@PASS,=V($NUCENTR)
0003CA                                       743   2          CNOP  2,4
                                             744   2        ##BALR  R@EXIT,R@PASS
0003CA 05 EF                                 745   3          BALR  R@EXIT,R@PASS
0003CC 00000060                              746   2          DC    A(96)
0003D0 E2F1404040404040                      747   2          DC    CL8'S1'
                                             748              PRINT NOGEN
0003D8                                       749              @DATA CLASS=S,BASE=R12,INIT=VAR
                                             754     * CALCULATE MATRIX POSITION
0003DC 58 20 1000                            755              L     R2,0(0,R1) ADDRESS OF FIRST PARAMETER
0003E0 48 70 2000                            756              LH    P,0(0,R2)    VALUE OF FIRST PARAMETER
0003E4 06 70                                 757              BCTR  P,0
0003E6 4C 70 C00A      00000142              758              MH    P,NUMCOLS
0003EA 58 20 1004                            759              L     R2,4(0,R1) ADDRESS OF SECOND PARAMETER
0003EE 4A 70 2000                            760              AH    P,0(0,R2)    VALUE OF SECOND PARAMETER
0003F2 06 70                                 761              BCTR  P,0
0003F4 1A 77                                 762              AR    P,P
0003F6 1A 77                                 763              AR    P,P
                                             764     * PERFORM FUNCTION
0003F8                                       765              @EXIT
000400                                       773              @END
                                             781              PRINT GEN
                                             782              EJECT
```

```
ASSEMBH LISTING                                                    14:29:44  1994-03-08   PAGE 0

LOCTN  OBJECT CODE    ADDR1    ADDR2     STMNT  M  SOURCE STATEMENT
                                           783     * ACCEPTANCE OF ADDRESSES FROM PARAMETER ADDRESS LIST
                                           784     *                           INTO ANOTHER ADDRESS LIST
                                           785     S2       @ENTR TYP=I,TITLE=NO,LOCAL=PARLI3,PASS=STA,PLIST=(ALN,ACN)
                                           786   1          @@SYN ,@ENTR,,EQ,0
000410                                     787   1 S2       DS    0D
000410                 00000000            788   1          USING @SAV,R@STACK
000410 90 EC D00C      0000000C            789   1          STM   R14,R12,@SAVR14
000414 18 AF                               790   1          LR    R@BASE,R@PASS
000416                 00000410            791   1          USING S2,R@BASE
                                           792   1          @PASS EXTNAME=$NUCENTR,CNOP=(0,4),DC=(A(LPARLI3),CL8'S2')
000416 58 F0 A050      00000460            793   2          L     R@PASS,=V($NUCENTR)
00041A                                     794   2          CNOP  2,4
                                           795   2        ##BALR  R@EXIT,R@PASS
00041A 05 EF                               796   3          BALR  R@EXIT,R@PASS
00041C 00000068                            797   2          DC    A(LPARLI3)
000420 E2F2404040404040                    798   2          DC    CL8'S2'
                                           799   1          @DATA BASE=R@STACK,DSECT=PARLI3
000428                 00000000            800   2          USING PARLI3,R@STACK
                                           801   1          @@PPP E,R@PAR,0,0,STA,(ALN,ACN)
000428 D2 03 D0641000 00000064             802   2          MVC   ALN(4),0(R@PAR)
00042E D2 03 D0601004 00000060             803   2          MVC   ACN(4),4(R@PAR)
                                           804              PRINT NOGEN
000434                                     805              @DATA CLASS=S,BASE=R12,INIT=VAR
                                           810     * CALCULATE MATRIX POSITION
000438 58 20 D064      00000064            811              L     R2,ALN    ADDRESS OF FIRST PARAMETER
00043C 48 70 2000                          812              LH    P,0(0,R2)    VALUE OF FIRST PARAMETER
000440 06 70                               813              BCTR  P,0
000442 4C 70 C00A      00000142            814              MH    P,NUMCOLS
000446 58 20 D060      00000060            815              L     R2,ACN    ADDRESS OF SECOND PARAMETER
00044A 4A 70 2000                          816              AH    P,0(0,R2)    VALUE OF SECOND PARAMETER
00044E 06 70                               817              BCTR  P,0
000450 1A 77                               818              AR    P,P
000452 1A 77                               819              AR    P,P
                                           820     * PERFORM FUNCTION
000454                                     821              @EXIT
000460                                     829              @END
                                           837              PRINT GEN
                                           838     PARLI3   @PAR  D=YES,LEND=YES,PLIST=(ACN,ALN)
                                           839   1          @@SYN ,@PAR,,LE,1
000000                                     840   1 PARLI3   DSECT
000000                 00000060            841   1          ORG   *+96
000060                                     842   1 ACN      DS    A
000064                                     843   1 ALN      DS    A
                       00000068            844   1 LPARLI3  EQU   *-PARLI3
000470                                     845   1 DEMOPARA CSECT
                                           846              EJECT
```

```
ASSEMBH LISTING                                                                14:29:44  1994-03-08   PAGE 0

LOCTN  OBJECT CODE    ADDR1    ADDR2     STMNT  M  SOURCE STATEMENT
                                           847     * ACCEPTANCE OF ADDRESSES FROM PARAMETER ADDRESS LIST
                                           848     *                            INTO REGISTERS
                                           849     S3       @ENTR TYP=I,TITLE=NO,PASS=STA,PLIST=((R2),(R1))
                                           850   1          @@SYN ,@ENTR,,EQ,0
000470                                     851   1 S3       DS    0D
000470                 00000000            852   1          USING @SAV,R@STACK
000470 90 EC D00C      0000000C            853   1          STM   R14,R12,@SAVR14
000474 18 AF                               854   1          LR    R@BASE,R@PASS
000476                 00000470            855   1          USING S3,R@BASE
                                           856   1          @PASS EXTNAME=$NUCENTR,CNOP=(0,4),DC=(A(96),CL8'S3')
000476 58 F0 A040      000004B0            857   2          L     R@PASS,=V($NUCENTR)
00047A                                     858   2          CNOP  2,4
                                           859   2        ##BALR  R@EXIT,R@PASS
00047A 05 EF                               860   3          BALR  R@EXIT,R@PASS
00047C 00000060                            861   2          DC    A(96)
000480 E2F3404040404040                    862   2          DC    CL8'S3'
                                           863   1          @@PPP E,R@PAR,0,0,STA,((R2),(R1))
000488 58 20 1000                          864   2          L     R2,0(0,R@PAR)
00048C 58 10 1004                          865   2          L     R1,4(0,R@PAR)
                                           866              PRINT NOGEN
000490                                     867              @DATA CLASS=S,BASE=R12,INIT=VAR
                                           872     * CALCULATE MATRIX POSITION
000494 48 70 2000                          873              LH    P,0(0,R2)     VALUE OF FIRST PARAMETER
000498 06 70                               874              BCTR  P,0
00049A 4C 70 C00A      00000142            875              MH    P,NUMCOLS
00049E 4A 70 1000                          876              AH    P,0(0,R1)     VALUE OF SECOND PARAMETER
0004A2 06 70                               877              BCTR  P,0
0004A4 1A 77                               878              AR    P,P
0004A6 1A 77                               879              AR    P,P
                                           880     * PERFORM FUNCTION
0004A8                                     881              @EXIT
0004B0                                     889              @END
                                           897              END


FLAGS IN 00000 STATEMENTS, 000 PRIVILEGED FLAGS, 000 MNOTES
HIGHEST ERROR-WEIGHT : NO ERRORS
THIS PROGRAM WAS ASSEMBLED BY ASSEMBH     V 1.2A00     ON 1994-03-08 AT 14:07:24
THIS LISTING WAS GENERATED BY THE LISTING GENERATOR V 1.1A00.
```

11.6 Differences between ASSEMBH V1.1A and ASSEMB V30.0A

The tables below list the functions and language elements where there are differences between ASSEMBH V1.2A and ASSEMB V30.0A. Functions not listed remain the same.

x Function is supported
– Function is not supported

	ASSEMBH V1.2A	ASSEMB V30.0A
SDF user interface	x	-
/PARAM command	-	x
Task switch control in DUET programming	-	x
Task switch control for the MCALL option	-	x
COMOPT interface	x	x
*COMOPT ADIAG=	x (not ASSEMBH-BC)	x
COPYMAC	-	x
ERRFIL	x	x
ISD	x (not ASSEMBH-BC) Creation of LSD information	Creation of ISD records
MCALL	-	x
MDIAG	-	x
OUTPUT	-	x
PROCOM	-	x
SAVLST	x (not ASSEMBH-BC)	x
SEQ	x	x
SOURCE = +	-	x
SYSPARM	max. 255 characters	max. 8 characters
UPD	-	x
	If unsupported options are used, ASSEMBH outputs a message	
Object code output		
in OM format	x	x
in LLM format	x	-

	ASSEMBH V1.2A	ASSEMB V30.0A
AID interface Storage of AID constant in object module	x (not ASSEMBH-BC) x If the TEST-SUPPORT=AID option is set, an 8-byte long consistency constant is stored in the object after the first program section	- -
IDA interface	-	x
ASSDIAG / ADIAG COMOPT, OPEN commands Explicit start CDT instructions CONTINUE-CDT command	x (not ASSEMBH-BC) - - x x	x x x - -
Number of possible MACRO and COPY libraries	100 each	5
ISLU	-	x
HALSTEAD metrics	-	x
Support for upper/lowercase	x	-
Maximum number of ESID numbers	$2^{31}-1$	2500
Maximum number of control sections	$2^{15}-1$	512
Assignment of statement number	All statements except macro instructions in macro generation and macro remarks are given a statement number if they contain errors or MTRAC is specified. A continuation line is given the number of the start line of the statement.	The statement number is assigned in accordance with PRINT control, error messages and MTRAC control. Each statement printed is given a separate number. In macro definitions, each continuation line receives a separate number.

Machine instructions, assembler instructions, remarks

	ASSEMBH V1.2A	ASSEMB V30.0A
Length of names External names Underscores in names	64 characters 8/32 characters x	8 characters 8 characters -
Names as elements in expressions	need not be defined beforehand. (exception: in the operand of the ORG instruction)	must be defined before being used in instructions
Arithmetic expressions	Any nesting of parentheses parenthesized	Up to 6 levels of parentheses only digits can exist without parentheses (error of the assembler)
Continuation lines	Any number	3
Machine instructions EXST LBF LWI POP PUSH STBF STWI ESA instructions	 - - - - - - - x	 x x x x x x x -
ISEQ instruction	- The instruction is treated as a remark	x
COPY statement	Nesting level, maximum 255	Nesting level, maximum 5
DC instruction	Duplication factor up to $2^{24}-1$	Duplication factor to $2^{16}-1$

	ASSEMBH V1.2A	ASSEMB V30.0A
EJECT statement	is listed before paper advance An MNOTE is issued in case of invalid operands	is not listed Invalid operands are ignored (error of the assembler)
EQU instruction	Negative values are possible	Only positive values
LTORG instruction	Any number in the source program	Maximum 255
Missing LTORG instruction	The literal pool is listed after the END statement	The literal pool is listed before the END statement
Invalid literals	are not included in the literal pool	are also included in the literal pool
Literal storage in the literal pool and literal XREF	Semantically identical but syntactically different literals are stored in the literal pool and literal XREF once for each literal definition. Example: Literals '=A(B)' and '=AL4(B)' are both stored. Syntactically identical literals are stored internally in the literal pool and XREF only once. The different forms of literal storage can cause differences in the object; in certain circumstances this can lead to REPs no longer being suitable.	Semantically identical but syntactically different literals are stored in the literal pool and literal XREF once only. For syntactically identical literals, the same applies as for ASSEMBH.

	ASSEMBH V1.2A	ASSEMB V30.0A
MNOTE instruction	A message is issued if the space between operand and remark is omitted	If the space is omitted, anything after the apostrophe is interpreted as a remark (error of the assembler)
OPSYN instruction OPSYN instruction, processing of model statements in macros	also inside macros For macro and COPY statements and REPRO instructions, the OPSYN in the macro definition applies. The OPSYN directly before it applies to all other instruction statements	only outside macros For library macros, the last OPSYN in the source applies; for macros in the source, the OPSYN prior to the definition applies
PRINT instruction Operand BASE CLOSED/OPEN DECK/NODECK NUM/NONUM NOBF/DBF REF/NOREF xON/xOFF	 The remark from the USING instruction is also output for each base address register - - - - - -	 Only the addressable area is output x x x x x x
PUNCH instruction REPRO instruction	No longer supported for module output in LLM format	no LLM format
SPACE statement	is listed before paper advance An MNOTE is issued if an operand is invalid	is not listed Invalid operands are ignored (error of the assembler)
TITLE statement	is listed before paper advance The name entry is not output to a PLAM object module	is listed after paper advance The name entry is always output to an object module

	ASSEMBH V1.2A	ASSEMB V30.0A
Register 0	A warning is output if a memory operand is accessed with base and index register 0	A warning is output if the assembler uses a value of 0 for the 2nd operand of an SS-type instruction and there is no base register
XDSEC instruction without operand or with invalid operand	R is assumed unless an external dummy of the same name is defined	An error message is issued
Remarks line with .* at the beginning of the assembler source program are	treated in the same way as a macro definition, and not printed out	not permitted in the assembler source program

Macro language elements

	ASSEMBH V1.2A	ASSEMB V30.0A
Macro definitions in the source program	The definition must be executed before the first instruction of the macro	The definition may be at any position in the source program
Redefinition of assembler instruction statements (in macro definitions in source program)	The mnemonic operation code of assembler instruction statements can be used as a macro name The macro concerned replaces the corresponding instruction statement	Redefinition of mnemonic operation code is possible only via the OPSYN instruction.
Generated mnemonic operation code of assembler instructions	The instructions ICTL, COPY, MNOTE and REPRO must not be generated	The instructions ICTL, START, COM, MNOTE, REPRO and COPY must not be generated
Conditional assembly	Macro definitions in the source program are only read in when they are executed Library macros are only read in when their call has been executed	Macro definitions in the source are always read in Library macros are are always read in if their macro call is in the source program
Inner macro definitions	A macro definition may be nested within another	No macro definition may lie within another
Macro definitions of the same name in the source program or in a macro definition	The last definition read in is valid until the next one	The first definition is valid; others are treated as errors

	ASSEMBH V1.2A	ASSEMB V30.0A
Remarks lines with .* at the beginning	are also possible before the prototype statement must not be generated	are only possible after the prototype statement must not be generated
Attribute references	yield valid attributes even if names not yet defined are referenced	For names still to be defined, the attributes are always undefined.
Definition attribute (D') reference	x	-
Number attribute reference (N')	yields the current dimension for SET symbols	is not allowed for SET symbols
Generated sequence symbols	are always recognized and processed	are only recognized if defined in a GSEQ instruction.
Sequence symbols in the name entry of generated instructions	are not listed	are listed
Sequence symbols defined more than once	Error message	Error message only if MTRAC is activated
Length of character values and substrings	Maximum 1020 characters	Maximum 255 characters
Concatenation of substrings	The concatenation point before the duplication factor of the 2nd string may be omitted	The concatenation point must be retained
Notation of character values	The notation C'...' is not allowed	C'...' is allowed. The C is ignored (error of the assembler)
Arithmetic macro expression	Character values not allowed as elements	Character values are converted. If not possible, no message is issued, and 0 is used in further computations (error of the assembler).

Macro instruction and prototype statement

	ASSEMBH V1.2A	ASSEMB V30.0A
Macro instruction and prototype statement	Positional and keyword operands may be specified in mixed sequence.	All positional operands must be given before all the keyword operands.
Inner macro instructions	Inner macro instructions are always resolved when they are executed.	If the MCALL option is specified, inner macro instructions are resolved only if defined in an MCALL instruction.
Location counter of the macro call	The current location counter is not listed.	The current location counter is listed.
Generated macro names	Generated macro names are always recognized and processed.	Generated macro names are only recognized and resolved if defined in an MCALL instruction.
Length of macro instruction operands	Maximum 1020 characters	Maximum 127 characters
Expressions in macro call operands	If the operand of a macro call is an expression, the corresponding symb. parameter has the attributes of a C-type constant.	If the operand of a macro call is an expression, the symb. parameter has the attributes of the 1st name in the expression.

Macro statements

	ASSEMBH V1.2A	ASSEMB V30.0A
ACTR instruction	The ACTR counter is preset to 4096	The ACTR counter is preset to 1200
AIF and AGO instructions	Extended format possible in both cases (see section 7.2) Alternative statement format possible in both cases	- Normal format only
LCLx and GBLx instructions	Alternative statement format possible in all cases	Normal format only
OPSYN instruction	Possible even within a macro definition	Only possible outside the macro definition
GSEQ instruction MCALL instruction	- - Both instructions are flagged with a warning, otherwise ignored	x x
Listing of hexadecimal and self-defining terms	are listed in decimal form	The original instruction is listed

Variable symbols

	ASSEMBH V1.2A	ASSEMB V30.0A
Generated parameter names	x	-
Implicit definition of local SET symbols	x	-
SETA symbols	If it has an arithmetic value, a SETC symbol may be assigned to a SETA symbol	Any character expression that has an arithmetic value may be assigned to a SETA symbol
SETB symbol	If its value is 0 or 1, a SETC symbol may be assigned to a SETB symbol	Not allowed
System variable symbol &SYSECT	is also reserved for names of COM and XDSEC	is only reserved for START, CSECT and DSECT
&SYSLIST(n) with n=0	produces the name entry of the macro instruction	is not allowed
&SYSPARM &SYSTSEC &SYSVERM/&SYSVERS	Max. 255 characters x are truncated to the right or padded to the right with _	Max. 8 characters - are truncated to the right or padded to the left with 0
System variable symbols in the operation entry	The operation entry can be generated with system variable symbols	System variable symbols are not allowed in the operation entry
Concatenation of variable symbols and alphanumeric characters	If a concatenation is used in assembler instructions, the concatenation point must be written.	In concatenations in assembler instructions, the concatenation point may be omitted (assembler error)

Structured programming with ASSEMBH

		ASSEMBH V1.2A	ASSEMB V30.0A COLUMBUS-ASSEMBLER V2.2F
STACK management		Dynamic STACK management. The STACK's logical structure complies with the PROSYS standard linkage convention	Static storage in memory
@ENTR macro operands	ENTRY=	x	-
	AMODE=	x	-
	RMODE=	x	-
	STACK=	x Size of the dynamically extendable initial stack No entry = one page (4096 bytes) STACK=n signifies an initial stack of n bytes	x Minimum stack, dynamically extendable (no STACK entry) or static, non-extendable stack of n bytes (STACK=n)
	KL5SP=	- A KL5SP entry is ignored with MNOTE 0	x
	ILCS=YES	x	-
	STREQ	x	-
	STREL	x	-
	HPREQ	x	-
	HPREL	x	-
	SLTERM	x	-
	SCTERM	x	-
	EXTMIN	x	-
	ABKR	x	-
	PROCHK	x	-
	ERROR	x	-
	OTHEVT	x	-
Entry IASSIN (COLBIN)		ignores a KL5SP specification	supports a KL5SP specification

	ASSEMBH V1.2A	ASSEMB V30.0A/ COLUMBUS-ASSEMBLER V2.2F
@CONDI macro @CONEN macro @EVTLC macro @EVTOE macro @ININ macro @SETJV macro @SETPM macro @STXDI macro @STXEN macro @STXIM macro	x x x x x x x x x x	- - - - - - - - - -
@END macro Operand DROP=	x additional operand value =(), all USING registers are released	x

12 Manual supplements

This chapter is an update for the present manual valid for ASSEMB V1.2D.

12.1 SPACE instruction

Section 4.2 Description of instructions SPACE Line feed (page 110)

Description

"no" modulo 100 specifies the number of blanks which are to be printed after the SPACE instruction in the assembler listing.

If "no" modulo 100 is greater than the number of lines remaining on this page, the SPACE instruction produces a page feed.

12.2 Variable system parameter

Section 6.3 Variable system parameter (page 176)

&SYSDATE_ISO4

Value of &SYSDATE_ISO4: yyyy-mm-dd Counter base is 10

The year is output in 4 digits, otherwise all as with &SYSDATE.

12.3 Type S procedures

Section 9.3.2.2 (page 266)

Type S procedures

As with procedure types B, D and L, type S procedures are not connected to the memory management. No new save area is provided and dynamic parameter passing is not possible. However, in contrast to types B, D and L, the register states of the calling procedure are saved in their save area.

The ENTRY= parameter can be used to control whether a CSECT or ENTRY statement is generated.

By default, register 15 is assigned as the base register. The user can assign some other register as the base register in the BASE operand of the @ENTR macro. This register must be loaded with the correct value at the start of the procedure (e.g. with the Assembler instruction "LR reg,R15").

Since the procedures are on a low level (no connection to the runtime system), the user has to restore the registers manually if required, i.e. the registers are not reloaded when the procedure terminates.

For the same reason, calling further subroutines with @PASS should be avoided, since tracing is not possible because of the missing save area chaining.

12.4 Procedure linking and parameter passing

Section 9.5 Procedure linking and parameter passing (page 277)

No procedure stack is provided when calling type B, L, D or S procedures.

12.5 Static parameter passing

Section 9.5.1.1 Static parameter passing (page 279)

This form of parameter passing is allowed in procedures of type M, E, I, L, B and S.

12.6 Contingency handling

Section 9.6.4 Contingency handling (page 298), second-last paragraph:

The standard contingency handler then calls the user contingency routine in accordance with ILCS conventions.

The user contingency routine receives the information on the interrupted process through R1.

R1 points to a parameter list with the following contents:

Word 1: Contingency message;
corresponds to the CONMSG parameter in the @CONEN macro (page 310)
(see also the section on 'Information transfer to contingency processes' in section 4.3.6, 'Contingency processes', of the 'BS2000 Executive Macros' manual.

Word 2: Event information code;
(see above manual, same section, Table 10, 'Event information codes')

Word 3: Pointer to post code;
(see above manual, same section)

12.7 STXIT handling

Section 9.6.5 STXIT handling (page 299), 3rd. paragraph:

All STXIT events defined in BS2000 are supported by ILCS. The TERM and ABEND events are only handled internally by ILCS and are not forwarded via @STXEN to a routine enabled by the user.

12.8 Predefined macros for structured programming

Chapter 10 Predefined macros for structured programming

– @CONEN Enable contingency routine (page 310), first paragraph:
This macro instruction is allowed only in procedures with ILCS=YES and will only work properly if the initialization with ILCS has been correctly performed.

– @CONEN Enable contingency routine (page 310), last paragraph:

Return codes in register 15:

0 Processing completed normally

1 Register 13 has an invalid value

2 ILCS is not correctly initialized

3 Invalid parameter value in CONLEV

All other return codes signal internal errors.

@ENTR (Typ=M/E/I), Parameter ILCS=YES (page 327, 335)
Full connection to ILCS, allows specifying the following parameters...

@ENTR (Typ=M/E/I), programming information (page 330, 336)

The abort identifier in the event handler list is required if the standard event handler has to abort the search for event handling routines. Non-ILCS procedures within the procedure nesting structure are recognized by ILCS and skipped (no event handler list is created for non-ILCS procedures).

@ENTR (Typ=E/I), parameter RETURNS=YES (page 333)

If ILCS=YES: registers 2 through 14 are reloaded.

@ENTR (Typ=B/L), parameter TYP=B (page 338)

Defines that this is a procedure that can be called from any module (external procedure) and is not connected to the memory management or register saving.

@ENTR with new procedure type S (page 340.1)

Format 5: Type S procedures

```
------------------------------------------------------------------
Name     | Operation | Operands
---------|-----------|--------------------------------------------
[name]   | @ENTR     | TYP=S
|        | [,BASE=reg],
|        | [,ENTRY= {ENTRY|CSECT
|        |                 [,AMODE={24|31|ANY}
|        |                  ,RMODE={24|ANY}]}]
------------------------------------------------------------------
```

name Name of the procedure

reg General purpose register; positive absolute expression, either:

- decimal self-defined value or
- predefined name of a register or
- name to which an appropriate self-defined value was assigned.

a) Description

This is a type S procedure, without connection to the runtime system memory management. In contrast to the type B, D and L procedures, the register states of the calling procedure are saved with this type of procedure.

BASE=reg Assigns a register to the procedure as a base register.

If this specification is omitted, register 15 is assigned as the base register by the Assembler.

ENTRY= Specifies which entry structures are to be generated for the procedure.

The following is generated if ENTRY is specified:

```
name1  DS     0D
ENTRY         name1
```

The following is generated if CSECT is specified:

```
name1         CSECT
```

AMODE= Allocates an addressing mode to the procedure (see 4.2, AMODE instruction)

RMODE= Allocates a load attribute to the procedure (see 4.2, RMODE instruction)

If an invalid combination of AMODE and RMODE is specified, an MNOTE is generated, and AMODE 24 and RMODE 24 are used for both values.

b) Programming information

1. The procedure is assigned register 15 (entry point) as the base address register by default, unless some other register was specified with BASE.
2. The procedure may not be the first in a module.
3. The procedure type is only permitted if ILCS=NO.

@ENTR (Typ=M), parameter ENV=C (page 327)

The restriction 'only permitted if ILCS=NO' has been eliminated.

@ENTR (Typ=E/I), parameter ENV=C (page 334)

The entry is only permitted for type E procedures.

@ENTR (Typ=E/I), parameter ENTRY= (page 334)

@ENTR with the ENTRY= parameter is permitted for type E procedures as well as for type I procedures.

If @ENTR TYP=I with the ENTRY=ENTRY and ENV=SPL... parameters is specified a base register is attached to the internal entry point.

@ENTR (Typ=E), programming information (page 336)

1. The abort identifier ...
2. @ENTR TYP=E with the ENV=C and ENTRY=ENTRY parameters only generates a secondary entry in the next higher procedure and not a new procedure. In contrast to procedures without an environment entry, this definition may not be terminated with @END.

@EXIT Format 1 (page 344)

– Format 1: Return from type M, B or L or S procedures

  In type B, L or S procedures, the program is continued with the instruction that follows the @PASS macro in the calling procedure.

@PASS programming information (page 361)

1. A procedure called with EXTNAME=ext_name must be defined with @ENTR TYP=E/B/S.

12.9 Dummy registers: Examples

Section 11.4 Dummy registers: Examples

In Example 1/Part 1 the declaration of PSRVEKT is not correct.

It must be changed to
PSRVEKT DS CL(L'PSREG1 + L'PSREG2 + 20 * L'PSREG3)

12.10 Differences between ASSEMBH V1.2A/ASSEMB V30.0A

Section 11.6 Differences between ASSEMBH V 1.2A/ASSEMB V30.0A

In the listing for DSECT no object code is printed even if a DC or instructions are contained in the DSECT.

References

[1] **ASSEMBH** (BS2000)
User Guide

[2] **AID** (BS2000)
Advanced Interactive Debugger
Debugging of ASSEMBH Programs
User Guide

[3] **Assembler Instructions** (BS2000/OSD)
Reference Manual

[4] **LMS** (BS2000)
SDF Format
User Guide

[5] **BS2000/OSD-BC**
DMS Introductory Guide
User Guide

[6] **BS2000/OSD-BC**
Executive Macros
User Guide

[7] **Introductory Guide to XS Programming (for Assembler Programmers)** (BS2000)
User's Guide

[8] **BS2000/OSD-BC**
Dynamic Binder Loader / Starter
User Guide

[9] **C** (BS2000)
C Compiler
User Guide

Index

B

C

E

F

G

M

Q

R

S

Contents

User Guide - English

FUJITSU

ASSEMBH

Reference Manual

Valid for
ASSEMBH V1.2
With Supplement chapter for ASSEMBH V1.2D

Edition June 2010

Comments… Suggestions… Corrections…

The User Documentation Department would like to know your opinion on this manual. Your feedback helps us to optimize our documentation to suit your individual needs.

Feel free to send us your comments by e-mail to:
manuals@ts.fujitsu.com

Certified documentation according to DIN EN ISO 9001:2000

To ensure a consistently high quality standard and user-friendliness, this documentation was created to meet the regulations of a quality management system which complies with the requirements of the standard DIN EN ISO 9001:2000.

cognitas. Gesellschaft für Technik-Dokumentation mbH
www.cognitas.de

Copyright and Trademarks

On April 1, 2009, Fujitsu became the sole owner of Fujitsu Siemens Computers. This new subsidiary of Fujitsu has been renamed Fujitsu Technology Solutions.
This document is a new edition of an earlier manual for a product version which was released a considerable time ago in which changes have been made to the subject matter.
Please note that all company references and copyrights in this document have been legally transferred to Fujitsu Technology Solutions.
Contact and support addresses will now be offered by Fujitsu Technology Solutions and have the format …@ts.fujitsu.com.
The Internet pages of Fujitsu Technology Solutions are available at http://ts.fujitsu.com/...