Part Two

80/AS Assembler Reference Guide

Table of Contents

1. Introduction

This portion of the Experts-PL/M™ manual describes the syntax of the 80/AS™ assembly language, and the operation of the 80/AS assembler. It is intended to be used primarily as a reference guide, and not as a tutorial. Some familiarity with assemblers in general is assumed, as well as an understanding of the 8080/8085 or Z80 microprocessor.

1.1 FEATURES AND CAPABILITIES

The 80/AS language is sufficiently similar to Intel's 8080/8085 Assembly Language that programs written for that assembler should assemble correctly using the 80/AS assembler with little or no modifications to the source.

The most significant area of change is in the handling of assembler controls. Instead of having special control lines beginning with a '$', 80/AS has assembler directives that appear in the operation field of statements. For a discussion of each assembler directive, see directives. A glossary of the directives and their formats is the subject of directivesA.

The other significant modifications or extensions are enumerated below.

- Optional use of names up to 31 characters in length. This allows direct access to all symbol names output by 80/PC.
- Common segments accessible for initialization and reference. Blank common and up to 249 named commons may be accessed, and their relocatabilities may be set to BYTE, PAGE, or INPAGE.
- Symbols with limited scope. Local symbols of the form #*name* only retain their definitions between the nearest preceding and following non-local labels. This provides a convenient way of generating labels for small loops, while at the same time minimizing the chance of multiple definition of symbols.
- Recognition of a limited number of Z80 instructions. The short relative jump (jr, jrc, jrnc, jrz, and jrnz) and word add and subtract (dadc and dsbb) instructions can be recognized and assembled correctly by 80/AS. The short jump instructions may also be translated into their 8080 equivalents, allowing the programmer to write a single routine that will take advantage of the shorter Z80 instructions, yet still be able to reassemble it and run it on an 8080 or 8085.

The assembler operates under the VAX/VMS™, UNIX™, and PC-DOS operating systems.

1.2 OBJECT MODULE FORMAT

The object module produced by 80/AS are in the Intel MCS-80/85 Relocatable Object Module Format.

2. Using the 80/AS Assembler

This chapter explains how to use 80/AS in the various operating system environments and discusses the various invocation options in detail.

2.1 INVOKING 80/AS UNDER UNIX AND PC-DOS

Under the UNIX and PC-DOS operating systems, the 80/AS Assembler is invoked by

```
80as [option...] source-file [source-file...]
```

80/AS requires one or more named files as input; if more than one input file is specified, the files are assembled separately. The name of each object file is derived from the input file name by stripping off any ".*" suffix, and postpending the ".q" suffix.

2.1.1 Invocation Options

The normal operation of 80/AS may be modified by several invocation options. These are

-c	Do not list source lines that were from the false branch of a conditional assembly. These lines are normally listed. Specifying the -c option has the same effect as placing the .NOCOND directive at the beginning of the source file.
-d	Generate local symbol records in the object file for use in debugging. These records are not normally produced.
-f	Generate a form feed (control L) at the end of each listing page. Whether a form feed or blank lines are generated by default is set at the time of installation.
-F	Generate the appropriate number of blank lines at the end of each listing page. Whether a form feed or blank lines are generated by default is set at the time of installation.
-g	Only list the generated code for macro expansions. This option limits the number of lines that appear in macro expansions. If a statement within a macro expansion does not generate any object bytes and the -g option is in effect, that statement will not be listed. Normally, all lines from macro

expansions are listed, so this option has no effect unless the -m option (don't list macro or repeat block expansions) is also specified.

-I*list* Specify a list of directories to search for include files; *list* is a colon-separated list of directories where include files are sought if not found in the directory of the source file. Multiple -I options may be specified. If an alternate search path is not specified, 80/AS will look in the directory of the source file, and then in the current directory, for any include files whose names are not rooted.

-l*length* Set the page length for listing to *length* lines. The length includes seven heading lines and three blank footing lines. If *length* is 0, the page length becomes infinite. In this case only the beginning of the listing, the beginning of the cross reference (if requested) and the beginning of the error listing (if any errors were detected) will cause a new page to be started. If this option is not used the installation default length is used for any listing output. Note that the use of the -l and -w options do not, by themselves, request the assembler to produce a listing and/or cross reference – see the -p and -x options.

-m Do not list macro expansions. Only the macro call will appear in the listing. Specifying the -m option has the same effect as placing the .NOGEN directive at the beginning of the source file. The default is to list macro expansions.

-n Do not generate an object file.

-p Produce a listing on the standard output. By default, no listing is generated. If either the width or the length is not also specified in the command tail, installation default values will be used.

-s Run in compatibility mode. All identifiers are silently truncated to 6 characters in length. Normally, identifiers may be up to 31 characters in length, simplifying the interface with programs written in 80/PL, but some existing assembly language programs may not assemble correctly with long identifiers enabled.

-w*width* Set the page width to *width* characters. If this option is not used, the installation default width is used for any listing output. Note that the use of the -l and -w options do not, by themselves, request the assembler to produce a listing and/or cross reference – see the -p and -x options.

-x Produce a cross reference after the source listing (if also requested) on the standard output. The page width and length are the same as for the listing. Normal action is to suppress the cross reference listing.

-z Map certain Z80 ops into 8080 equivalents. The 80/AS assembler recognizes a few Z80 ops (jr, jrz, jrnz, jrc, jrnc, dadc, dsbb) and generates Z80 instructions for them (These are the Z80 instructions that may be output by 80/PC when its -Z option is specified). The 80/AS -z option will translate the short relative Z80 jumps into their 8080 equivalents – the word add and subtract instructions have no 8080 single instruction equivalents, and an error message will be issued if either is encountered.

-Z Do not treat the Z80 ops as instructions. If your code contains user-defined symbols that conflict with the Z80 instructions (jr, jrz, jrnz, jrc, jrnc, dadc,

	dsbb), you will need to use this option.
-B*string*	Prepend *string* to the name of each assembler phase before executing it, thus allowing alternate versions of the assembler to be executed. This option may not be supported on all versions of 80/AS.
-Xs*aaa*	Specifies, as *aaa*, the default suffix to use for source file names that are not given with a suffix.
-Xo*aaa*	Specifies, as *aaa*, the suffix to be used on object files in place of the default ".q".
-Xl*aaa*	Specifies that any listing produced will be directed to a file, instead of to the standard output. The file will have the same name as the corresponding source file, but with a suffix of *aaa*.
-Xt*aaa*	Specifies, as *aaa*, the *prefix* to be used on all temporary file names, instead of the default "\tmp\" under PC-DOS or "/usr/tmp/" under UNIX. As an example, "-Xt./" will cause temporary files to be created in the current directory (i.e., the one in use when 80/AS is invoked).

2.1.2 Argument Concatenation

80/AS options may be concatenated, provided that any option that requires a non-numeric operand (e.g. -l*list*) is not followed by any other option. For example, the invocations

```
        80as -p -l66 -w79 -x -I/usr/include test.s
```

and

```
        80as -pl66w79xI/usr/include test.s
```

are equivalent.

2.1.3 Argument Files

Any command line argument may have the form

```
        @argfile
```

where *argfile* is a file containing more arguments. This is particularly useful in cases where more arguments are required than will fit on the original command line.

2.1.4 Redirecting the Standard Error File

Error messages are written on the standard error file, which is usually the display screen. This may be changed by using a command line (or argument file) argument of the form

```
        ^errfile
```

where *errfile* is the name of the file to receive error messages. If the argument has the form

```
        ^^errfile
```

the messages will be appended to the file.

2.1.5 Invocation Examples

The command

```
        80as -pxF -w106 -166 ctest.s > ctest.l
```

will assemble the file *ctest.s*, putting the object into *ctest.q*, and generating a listing (*p*) and cross reference (*x*) with a page width of 106 characters (*w106*) and page length of 66 lines (*l66*). Each page will be filled out with blank lines (*F*) rather than with a form feed, and the entire listing will be output to *ctest.l* (redirection of standard output).

2.2 RETURN CODES

Several different values may be returned by 80/AS, either because of some error detected by the assembler, or because of a '.PRINT' directive in the source program (see Section 4.9). The value returned by 80/AS indicates the most severe error it encountered during operation. The possible return values caused by assembler-detected errors are 0 - 4. As the return value gets larger, so does the severity of the indicated error.

The interpretation of error levels is as follows:

0	*Informative.* No error was detected.
1	*Warning.* An error was detected, but the assembler could still process the statement; however, the result might not be what the programmer intended.
2	*Error.* An error was detected, and the assembler had to abandon the statement. Most errors will be of this type. If the statement appeared to be an instruction, three null bytes (8080 NOP instruction) are inserted into the object module, to allow patching at execution time.
3	*Severe Error.* An error from which the assembler could not recover was encountered, causing the assembly to terminate. An example of this type of error is dynamic memory overflow.
4	*Fatal Error.* Errors of this type are internal assembler errors, and should not occur in normal operation.

See Appendix A for a list of error messages.

2.3 INVOKING 80/AS UNDER VMS

Under the VMS operating system, the 80/AS assembler is invoked by:

```
80AS [options] source-file-name

Command Qualifiers:              Defaults:

/[NO]CROSS_REFERENCE               /NOCROSS_REFERENCE
/[NO]DEBUG=(options)               /NODEBUG
/[NO]FORM_FEED                     /FORM_FEED
/[NO]IGNORE_Z80                    /NOIGNORE_Z80
/INCLUDES=(directory,...)
/LENGTH=len                        /LENGTH=66
/[NO]LIST[=file-spec]              /NOLIST
/[NO]MAP_Z80                       /NOMAP_Z80
/[NO]OBJECT[=file-spec]            /OBJECT
/SHOW=(options)                    /SHOW=(CONDITIONALS,
                                      EXPANSIONS)
/[NO]SHORT_NAMES                   /NOSHORT_NAMES
/WIDTH=wid                         /WIDTH=106
```

The normal assembler operation is assemble the specified file and place the output in a file with the same name and an extension of "Q80". The default extension for the source file is "A80". If a listing file is produced, it will by default have the same name as the source file with an extension of ".LIS".

The object files are, in general, not immediately executable. They should be ultimately linked with any required libraries and then bound to addresses reasonable for the final environment of the executable program.

The normal operation of the assembler may be modified by the use of various options as described in the following section.

2.3.1 Invocation Options

/CROSS_REFERENCE
/NOCROSS_REFERENCE

Controls whether or not a cross-reference listing will be generated. If so, it will appear at the end of the listing file. The default is /NOCROSS_REFERENCE.

/DEBUG
/NODEBUG

Controls whether the assembler generates local symbol records in the object file for possible use by a run-time debugger. The default is /NODEBUG.

/FORM_FEED
/NOFORM_FEED

Controls whether the assembler ends each listing page with a form feed character (^L) or the appropriate number of blank lines. The default is /FORM_FEED.

/IGNORE_Z80
/NOIGNORE_Z80

Controls whether the assembler recognizes certain Z80 instruction mnemonics (*jr, jrz, jrnz, jrc, dadc, dsbb*) or treats those symbols as other user-defined symbols. The default is /NOIGNORE_Z80.

/INCLUDES=(directory,...)

Specify directories to be searched for an INCLUDE file if the file is not found in the directory of the source file. The directories are searched in the order given.

/LENGTH=len

Sets the length of a page in the listing file, where len is the maximum number of lines on a page, including the heading and footing. If *len* is zero, no pagination is performed, except between the body of the listing, the error messages, and the cross-reference (if requested). The default is /LENGTH=66.

/LIST[=file-spec]
/NOLIST

By default, the assembler does not produce a listing if run interactively. If /LIST is specified, or the assembler is run in batch mode, the assembler produces a source listing file with the same name as the input source file but with a file type of "LIS". This may be overridden by giving a *file-spec.*

/MAP_Z80
/NOMAP_Z80

Controls whether the assembler maps (translates) Z80 short relative jumps into their 8080/8085 equivalents. The default is /NOMAP_Z80.

/OBJECT[=file-spec]
/NOOBJECT

Controls whether or not the assembler produces an object module. The default is /OBJECT which produces an object model that has the same file name as the source file and a file type of "Q80".

/SHOW=(options)

Controls whether certain kinds of source and generated code appear in the listing file. The following options are available:

[NO]CONDITIONALS	List source lines that are within conditional assembly blocks that were not assembled.
[NO]EXPANSIONS	List macro and repeat-block expansions.
[NO]BINARY	Only list macro and repeat-block expansions if they generate object. The BINARY option

is only meaningful when combined with the NOEXPANSIONS option.

The default is /SHOW=(CONDITIONALS,EXPANSIONS).

/[NO]SHORT_NAMES

Controls whether the assembler truncates all symbols to 6 characters in length. /NOSHORT_NAMES allows symbols to be up to 31 characters in length. The default is /NOSHORT_NAMES.

/WIDTH=wid

Sets the width of a listing page, where *wid* is the maximum number of characters on a line. The default is /WIDTH=106.

2.4 COMPLETION STATUS

Several different values may be returned by 80/AS, either because of some error detected by the assembler, or because of a '.PRINT' directive in the source program (see Section 4.9). The value returned by 80/AS indicates the most severe error it encountered during operation. The possible completion status values are:

Success	No error was detected.
Warning	An error was detected, but the assembler could still process the statement; however, the result might not be what the programmer intended.
Error	An error was detected, and the assembler had to abandon the statement. Most errors will be of this type. If the statement appeared to be an instruction, three null bytes (8080 NOP instruction) are inserted into the object module, to allow patching at execution time.
Severe	An error from which the assembler could not recover was encountered, causing the assembly to terminate. An example of this type of error is dynamic memory overflow.
Fatal	Errors of this type are internal assembler errors, and should not occur in normal operation.

See Appendix A for a list of error messages.

3. 80/AS Syntax

This chapter describes the syntax of 80/AS, including a discussion of expressions.

3.1 SYMBOLS

Variables in 80/AS are called *symbols*. Symbols may be from 1 to 31 characters in length, composed of alphabetic characters, numeric characters, and the characters '?', '_', and '@'. An upper case letter and its lower case counterpart are considered to be the same letter for purposes of symbol matching. Symbols must not begin with a numeric character.

3.1.1 Local Symbols

Symbols beginning with a '#' are treated specially. If a symbol begins with the '#' character, its definition and value are only known between the last preceding non-local label (or start of program) and the first following non-local label (or end of program). This symbol may be redefined outside its local block without causing a multiple definition error, and will not appear in the cross reference or local symbol records. Labels of this type are useful in constructing small loops without the necessity of creating a unique name for the target.

3.1.2 Generated Symbols

The assembler generates replacement names of the form "??nnnn", where *nnnn* is 0000, 0001, 0002, etc., in response to the LOCAL directive (Section 5.4). The programmer should therefore avoid using symbols of that form.

3.2 STATEMENT SYNTAX

An 80/AS program is composed of a series of statements. Each statement is terminated by a newline and may not be continued. A statement consists of up to four fields, separated by one or more blanks or tabs. The four fields, in order, are the *label* field, the *operation* field, the *operand* field, and the *comment* field:

```
        label:    operation        operand        ; comment
```

3.2.1 Label Field

The label field begins the statement. A label is a symbol immediately followed by a colon. It may be preceded by blanks or tabs. Many directives and all instructions may be labelled. This causes the symbol to be given the value of the current location counter *before* processing the rest of the statement. The relocatability of the symbol is that of the current segment. A statement may consist solely of a label.

3.2.1.1 Names

The SET, EQU, and MACRO directives require a *name* in the label field. A name is a symbol not followed by a colon. The symbol is given a value and type that depend on the specific directive. A single statement may not contain both a name and a label.

3.2.2 Operation Field

The operation field follows the label field. If there is no name or label, the operation field may begin the line. The operation field contains either an Intel 8080/8085 instruction mnemonic or an 80/AS assembler directive. Section 3.5 discusses the various formats when the operation is an opcode mnemonic. The format for each directive is given in Chapter 4.

3.2.3 Operand Field

The operand field follows the operation field. The specific operation determines which operand format or formats are required. This field generally continues until either the end of the line is reached, or until the comment field is encountered.

3.2.4 Comment Field

The comment field begins with a semicolon and continues until the end of the input line. This field is ignored by the assembler, and may contain any printable character, as well as blanks and tabs. A statement may consist solely of a comment.

3.3 VALUES

Many instructions and directives require an operand that represents a value. Each value is maintained within the assembler in two parts, a number and a relocatability. The numeric part of the value is a 16-bit unsigned quantity. The relocatability indicates the segment with which the value is associated.

For example, the constant 10 has a numeric value of 10, and its relocatability is absolute (it is not relocatable). A label that appears at the beginning of the code segment has a numeric value of 0, and its relocatability is the code segment.

The different ways that values may be represented are discussed individually below.

3.3.1 Symbols

A user symbol represents a value (unless it is the name of a macro). A symbol is given a value and relocatability either by use of the SET or EQU directives (see Chapter 4), or by its appearance in the label field of a statement (see Section 3.2.1). Symbols that have not yet been assigned a value are called undefined symbols; their value is taken to be absolute 0. In most contexts, the value of a symbol need not be known until the second pass, so forward referencing is allowed. For example, a jump instruction may refer to a label that appears later in the program. Restrictions on forward referencing are discussed in Section 4.25.

3.3.2 Constants

Constants are absolute quantities; they have no relocatability. They can be given in several different formats. Examples of each are presented below.

3.3.2.1 Numeric Constants

The assembler considers any token that begins with a digit to be a numeric constant. Numbers that end with a digit or the letter 'D' or 'd' are interpreted as base 10 values. Similarly, a terminal 'B' or 'b' implies a binary (base 2) number; a terminal 'O', 'o', 'Q', or 'q' implies an octal (base 8) number; and a terminal 'H' or 'h' implies a hexadecimal number (base 16). The letters A-F when used in a hex value may be either upper or lower case.

For example,

```
255 = 11111111B = 0377Q = 0ffh
```

but

```
FFH
```

is not a number, since it does not begin with a numeric character. Rather, it is the symbol "FFH".

3.3.2.2 Character Constants

Character constants are made up of 0 or more characters enclosed within single quotes ('). In order to include a quote within the string, use two successive quotes.

```
'Today''s date' ==> Today's date
```

A character string of length ≤ 2 may be used wherever a numeric value is expected. A string of length 0 has value 0; a string of length 1 has a value equal to the character's ASCII value; a string of length 2 represents a value whose high byte is the first character's ASCII value and whose low byte is the second character's ASCII value.

For example

```
''       ==>   0
'A'      ==>   65 = 41H
'ab'     ==>   24930 = 6162H
'abc'    ==>   (cannot be represented as a value)
```

3.3.2.3 Opcodes as Constants

A complete instruction enclosed within parentheses evaluates to a byte constant whose value is the first byte of object that the instruction would generate.

For example

```
(JMP     START)
```

has the value 0C3H.

3.3.3 Register Names

The seven byte registers have the predefined names A (accumulator), B, C, D, E, H, and L. The four word registers, three of which are composed of pairs of byte registers, have the predefined names SP (stack pointer), B (B and C registers), D (D and E), and H (H and L). In addition, some instructions can use the contents of the HL register pair as the address in memory of the actual operand. In this case the HL register pair is called M. Both upper and lower case register names are recognized. Whether a register name represents a single byte register or a register pair is determined by the context in which it is used. For example, the instruction

```
        MOV         A,B
```

moves the contents of the byte register B to register A, while

```
        PUSH        B
```

pushes the contents of the register pair BC onto the stack.

3.4 EXPRESSIONS

The assembler can perform assembly-time arithmetic on both absolute and relocatable values, resulting in a value that may be either absolute or relocatable, depending on both the operator(s) and operand(s). Internally, all values are held as 16-bit unsigned quantities, and arithmetic is unsigned modulo 10000H. Operators that yield a logical (true/false) result produce 0FFFFH for true, and 0 for false.

3.4.1 Order of Evaluation

The order in which expressions with more than one operator are evaluated depends upon the relative precedence of the operators. Terms containing operators with high precedence are evaluated before those containing operators with a lower precedence. If two terms have operators with the same precedence, the leftmost term is evaluated first. The precedence of the various operators is:

highest	parenthesized expression
	NUL
	SHL, SHR, MOD, *, /
	+(unary and binary), -(unary and binary)
	LT, LE, EQ, NE, GT, GE
	NOT
	AND
lowest	OR, XOR

3.4.2 Logical operators – NOT, AND, OR, XOR

The AND, OR and XOR (exclusive or) operators take two 16-bit absolute value operands and produce a 16-bit absolute result. The unary operator NOT produces the 1's complement of its 16-bit absolute operand.

```
01110111B  AND  00001101B    ==>    00000101B
01110111B   OR  00001101B    ==>    01111111B
01110111B  XOR  00001101B    ==>    01111010B
           NOT  00001101B    ==>    11110010B
```

3.4.3 Relational operators – LT, LE, EQ, NE, GT, GE

The relational operators compare two absolute or relocatable values. Relocatable operands must have the same relocatability. The result of the comparison is either true (0FFFFH) or false (0).

LT	(less than)	1	LT	0	==>	false
LE	(less than or equal)	1	LE	0	==>	false
EQ	(equal)	1	EQ	0	==>	false
NE	(not equal)	1	NE	0	==>	true
GT	(greater than)	1	GT	0	==>	true
GE	(greater than or equal)	1	GE	0	==>	true

3.4.4 Arithmetic Operators – MOD, +, –, *, /

There are five binary and two unary arithmetic operators:

"+" The binary "+" operator can take either two absolute values or an absolute and a relocatable value as operands. If one operand is relocatable, the result has that relocatability. Otherwise, the result is absolute.

The unary "+" operator is provided for completeness. The result is the same as the operand.

"-" The binary "-" operator takes (a) two absolute values, yielding an absolute value; (b) an absolute value subtracted from a relocatable value, yielding a result with the same relocatability as the subtrahend; or (c) two relocatable quantities, with the same relocatability, yielding an absolute result.

The unary "-" operator returns the 2's complement of its absolute operand.

"*" The "*" operator is the multiplication operator. Both operands must be absolute values.

"/" The "/" operator is the division operator. Both operands must be absolute values. The result of division by 0 is undefined.

MOD The MOD operator is the modulus operator. It returns the remainder of division of the first operand by the second operand. Both operands must be absolute values. The result of (number MOD 0) is undefined.

3.4.5 Shift operators – SHL, SHR

The shift operators require two absolute operands. The first is shifted left or right by the number of bit positions given by the second operand. A right shift fills the high bit with 0, while a left shift fills the low bit with 0.

```
11111111B  SHL  2  ==>  11111100B  (shift left)
11111111B  SHR  2  ==>  00111111B  (shift right)
```

3.4.6 The NUL operator

This operator is generally used to test for the presence or absence of a macro parameter. If the rest of the line following the NUL operator is blank (except for a comment), the operator returns true (OFFFFH); otherwise the result is false (0).

An example of the use of the NUL operator is provided in Chapter 5.

3.5 INSTRUCTION FORMATS

There are five different formats for instructions, differing only in the operand field (any instruction may have a label and/or a comment field). Each is described below, with an example.

3.5.1 No Operands

The mnemonic itself contains all the information required by the assembler to generate the complete instruction.

Format

```
[label]    op
```

Example

```
RET
```

3.5.2 One Register Operand

The single operand is either a register or a register pair. The specific instruction determines how the operand is interpreted.

Format

```
[label]    op    reg
```

Example

```
POP     B
```

pops the top two bytes off the stack into registers B and C, while

```
ADD     B
```

adds the contents of byte register B to the accumulator.

3.5.3 Two Register Operands

The only instruction to use this format is MOV. The destination specifier precedes the source specifier.

Format

```
[label]    op    reg1,reg2
```

Example

```
MOV     A,C
```

moves the contents of byte register C to byte register A, and

```
MOV     A,M
```

moves the byte stored in memory at the address contained in the HL register pair to register A.

3.5.4 One Expression

The operand is evaluated as an expression (see Section 3.4). Some instructions require that the value of the expression be a byte value; the high byte of the result must be either all zeroes or all ones, giving a range of values from -256 (0FF00H) to +255 (00FFH).

Format

```
        [label]    op    expr
```

Example

```
        ORI     01110111B
```

will OR the 8-bit value 77H with the contents of the accumulator.

3.5.5 One Register and One Expression

The first operand is either a register or a register pair. The second is an expression, which may be either an 8-bit or a 16-bit value, depending on the instruction.

Format

```
        [label]    op    reg,expr
```

Example

```
        MVI     B,10
```

loads the byte register B with the value 10, and

```
        LXI     B,START
```

loads the register pair B and C with the value of START.

4. Assembler Directives

This chapter describes each of the directives, or *pseudo-operations*, which may be used with the 80/AS assembler.

4.1 .COND

The .COND directive sets the "cond" listing control switch to *yes*. Using the conditional assembly features of 80/AS, some input lines may be skipped by the assembler. When the "cond" switch is set to *no*, these lines do not appear in the listing. The default setting for the "cond" switch is *yes*.

The line containing the .COND directive never appears in the listing.

Format

```
        [label]     .COND
```

Notes

1. If the current setting of the "list" listing control switch is *no*, listing is suppressed, and setting the "cond" switch will have no effect.

See Also

.NOCOND (Section 4.6)

4.2 .EJECT

The .EJECT directive causes the current page to be ended and the heading for a new page to be printed. The current page is ended either with a form feed or with the appropriate number of blank lines needed to fill out the page, depending on the invocation option (see Section 2.1.1 and Section 2.3.1).

The line containing the .EJECT directive never appears in the listing.

Format

```
    [label]    .EJECT
```

Notes

1. If the "list" switch is *no*, the .EJECT is ignored.

4.3 .GEN

The .GEN directive sets the "gen" listing control switch to *yes*; this switch controls whether macro expansions will be listed. When set to no, only the macro calls are shown; when set to *yes*, complete macro expansions are listed. The default setting for the "gen" switch is *yes*.

The line containing the .GEN directive is never listed.

Format

```
        [label]     .GEN
```

Notes

1. If the "list" switch is set to no, the setting of the "gen" switch is irrelevant.

See Also

.NOGEN (Section 4.7)

4.4 .INCLUDE

The .INCLUDE directive causes a new input file to be opened. Lines are read from this new file and processed just as if they had come from the original source file. When the end of the new file is reached, processing of the original source file resumes with the statement following the .INCLUDE statement. The operand of the .INCLUDE directive is a quoted string; this is the name of the include file.

A rooted file name is assumed to be the complete path name of the include file. Otherwise, the file is searched for in the directory of the original source file. An alternate list of directories to search may be given when 80/AS is invoked. Invoke 80/AS with the -I*list* option (or the /INCLUDES=(list) qualifier). The directories may be either rooted or relative to the current directory.

A .INCLUDE directive may appear within an include file; the entire process recurses.

Format

```
        [label]    .INCLUDE    'file-name'
```

Examples

Under UNIX or DOS, to include the file "/usr/users/dave/includes.d/io.h", use

```
        .INCLUDE    '/usr/users/dave/includes.d/io.h'
```

Under VMS, the file "[users.test.include]io.h" can be included by

```
        .INCLUDE    '[users.test.include]io.h'
```

To include the file "io.h" from the current directory, use

```
        .INCLUDE    'io.h'
```

If a collection of commonly-referenced include files is in the directory "/usr/progA/include" and 80/AS is invoked with a *-I:/usr/progA/include* switch, then the directive

```
        .INCLUDE    'defs.h'
```

will cause the assembler to look first for "defs.h" in the current directory (the null directory name preceding the colon is equivalent to "./"), and then to look for "/usr/progA/include/defs.h".

4.5 .LIST

80/AS is capable of producing a formatted listing of the input source file and generated object on the standard output. The "list" switch controls whether a given source line and its associated generated object will be listed or not. The .LIST directive sets the "list" listing control switch to *yes*. When set to *yes*, listing is enabled. The default setting for the "list" switch is *no*; invoking 80/AS with the -*p* option (/LIST qualifier) sets the "list" switch to *yes*.

The line containing the .LIST directive is not printed.

Format

```
        [label]     .LIST
```

See Also

.NOLIST (Section 4.8)

4.6 .NOCOND

The .NOCOND directive sets the "cond" listing control switch to *no*. Using the conditional assembly features of 80/AS, some input lines may be skipped over by the assembler. When the "cond" switch is set to *no*, lines that are within unprocessed conditional assembly blocks are not listed. The conditional assembly directives themselves are also not listed. The default setting for the "cond" switch is *yes*; invoking 80/AS with the -*c* option (/SHOW=NOCONDITIONALS qualifier) sets the "cond" switch to *no*.

The line containing the .NOCOND directive never appears in the listing.

Format

```
        [label]     .NOCOND
```

Notes

1. If the current setting of the "list" listing control switch is no, listing is suppressed, and setting the "cond" switch will have no effect.

See Also

.COND (Section 4.1)

4.7 .NOGEN

The .NOGEN directive sets the "gen" listing control switch to *no*; this switch controls whether macro expansions will be listed. When set to *no*, only the macro calls are shown; when set to *yes*, complete macro expansions are listed. The default setting for the "gen" switch is *yes*; invoking 80/AS with the *-m* option (/SHOW=NOEXPANSIONS qualifier) sets "gen" to *no*.

The line containing the .NOGEN directive is never listed.

Format

```
        [label]     .NOGEN
```

Notes

1. If the "list" switch is set to *no*, the setting of the "gen" switch is irrelevant.

See Also

.GEN (Section 4.3)

4.8 .NOLIST

The .NOLIST directive sets the "list" listing control switch to *no*. The "list" switch controls whether a given source line and its associated generated object will be listed or not. When set to no, lines are not listed and .EJECT directives have no effect.

The line containing the .NOLIST directive is never listed.

Format

```
      [label]    .NOLIST
```

See Also

.LIST (Section 4.5)

4.9 .PRINT

The .PRINT directive causes a user-supplied error message to be printed on the standard error and entered in the error log printed at the end of the listing (if a listing was requested). An error level is associated with the message, allowing premature termination of the assembly. The assembler returns values of 0 (informative - no errors) through 4 (fatal error), but any value may appear in the .PRINT statement. However, any value greater than 2 (error) will cause immediate termination of the assembly. See the discussion of error levels in Section 2.2 and Section 2.4 for more discussion of return codes.

The .PRINT directive itself will not appear in the listing.

Format

```
[label]   .PRINT   expression, 'message'
```

Example

The .PRINT directive is especially useful when coupled with conditional assembly within macros. If a necessary parameter is not provided, or a value is outside its allowable range, an error can be printed.

```
   .
   .
IF       NUL      ARG1
.PRINT   2, 'Arg1 required but missing'
ELSEIF   ARG1 GE 256
.PRINT   2, 'Arg1 must be less than 256'
ENDIF
   .
   .
```

4.10 .RESTORE

The .RESTORE directive restores the settings of the "list", "gen", and "cond" listing switches that have been saved with a .SAVE directive (Section 4.11).

The line containing the .RESTORE directive does not appear in the listing.

Format

```
        [label]     .RESTORE
```

See Also

.SAVE (Section 4.11)

4.11 .SAVE

The .SAVE directive saves the current settings of the "list", "gen", and "cond" listing switches on a stack. A subsequent .RESTORE directive (Section 4.10) will restore the settings of the listing switches.

The save/restore mechanism is useful when writing macros that may be called from different places within the program, with potentially different listing switches in effect. At the beginning of the macro expansion, the switches are saved and altered appropriately. Just before the macro expansion ends the switches are restored to their previous values.

The .SAVE directive is not listed.

Format

```
    [label]    .SAVE
```

See Also

.RESTORE (Section 4.10)

4.12 .SEG

The .SEG directive allows the programmer to access the stack (seg id 3), memory (seg id 4), and unnamed common (seg id 255) segments, and to define up to 249 named common segments. The assembler saves the current location counter for the old segment, and sets it to the last value it had in the new segment. If the segment has not yet been referenced, the location counter is set to 0. The segment relocatability type may also be specified; each SEG directive referencing a given segment must specify the same segment relocatability type.

Format

```
        [label]    SEG    seg-name [, type-spec]
```

where *type-spec* is either PAGE or INPAGE. If the segment relocatability type is not specified, the relocatability type will be BYTE.

Seg-name is one of

//	the segment is set to unnamed common
STACK	the segment is set to the stack segment
MEMORY	the segment is set to the memory segment
other name	if the name is undefined, a new common segment is created with the given name. If the name has already been defined as a common segment, that becomes the new segment; otherwise the statement is in error.

See Also

ASEG directive (Section 4.15),
CSEG directive (Section 4.16),
DSEG directive (Section 4.19)

4.13 .SUBTITLE

The .SUBTITLE directive allows the programmer to set and change the subtitle that appears in the heading for each listing page. The .SUBTITLE directive may be used as often as desired. If a new page is to have a new subtitle, be sure to place the .SUBTITLE directive *before* the .EJECT directive, since the title and subtitle in effect at the time of the eject will be used for the new page heading.

The .SUBTITLE directive is not listed.

Format

```
        [label]     .SUBTITLE     'subtitle string'
```

The operand is a quoted string. If the string is the null string, the current subtitle will be deleted.

Notes

1. The first subtitle encountered will appear on the first page, and on all following pages, until after the next subtitle directive is encountered.
2. In the error listing and cross reference listing, the space occupied by the subtitle is occupied by the "ERROR LISTING" and "CROSS REFERENCE" strings, and so is not available to the programmer.

4.14 .TITLE

The .TITLE directive allows the programmer to set and change the title that appears in the heading for each listing page. The .TITLE directive may be used as often as desired. If a new page is to have a new title, be sure to place the .TITLE directive *before* the .EJECT directive, since the title and subtitle in effect at the time of the eject will be used for the new page heading.

The .TITLE directive is not listed.

Format

```
    [label]    .TITLE    'title string'
```

The operand is a quoted string. If the string is the null string, the current title will be deleted.

Notes

1. The first title encountered will appear on the first page, and on all following pages, until after the next title directive is encountered.

4.15 ASEG

The ASEG directive sets the current segment to be the absolute segment. The assembler saves the current location counter for the old segment, and sets it to the last value it had in the absolute segment. By default, object is placed in the absolute segment, beginning at location 0.

Format

```
        [label]    ASEG
```

See Also

.SEG directive (Section 4.12),
CSEG directive (Section 4.16),
DSEG directive (Section 4.19)

4.16 CSEG

The CSEG directive sets the current segment to be the code segment (seg id 1). The assembler saves the current location counter for the old segment, and sets it to the last value it had in the code segment. The segment relocatability type may also be specified; each CSEG directive must specify the same segment relocatability type.

Format

```
        [label]    CSEG [, type-spec]
```

where *type-spec* is either PAGE or INPAGE. If the segment relocatability type is not specified, the relocatability type will be BYTE.

See Also

.SEG directive (Section 4.12),
ASEG directive (Section 4.15),
DSEG directive (Section 4.19)

4.17 DB

The DB directive is used to put byte expressions and quoted strings into the object module. The expression may be absolute, relocatable or external, but all terms must have been defined when the expression is evaluated in the second pass.

Format

```
        [label]    DB    expr-or-string[,expr-or-string ...]
```

expr-or-string is either an expression that evaluates to a byte value (the high byte of the value is either 0 or 0FFH), or a quoted string. If more than one expression or quoted string appears in a DB directive they must be separated by commas.

Example

A program needs to print a signon message when it starts up. The output routine expects all strings to end with a null byte. A possible DB statement for the signon message is

```
                 .
        CR       EQU     0DH
        LF       EQU     0AH
                 .
        SIGNON:  DB      'Welcome to Mars!', CR, LF, 0
                 .
```

Notes

1. The current location counter is incremented after processing each member of the list of expressions or quoted strings. This means that the value of the "$" term changes from expression to expression when there is a list of expressions or quoted strings.

See Also

DW directive (Section 4.20)

4.18 DS

The DS directive increments the current location counter by the value of its operand, effectively allocating uninitialized space. The operand of DS must be an expression that evaluates to an absolute value; since the location of all labels must be determined by the end of the first pass, the increment expression must be defined when it is first encountered.

Format

```
        [label]    DS    expr
```

Example

To allocate a 128 byte buffer:

```
                   .
        INBUF:   DS       128
                   .
```

See Also

ORG directive (Section 4.29)

4.19 DSEG

The DSEG directive sets the current segment to be the data segment (seg id 2). The assembler saves the current location counter for the old segment, and sets it to the last value it had in the data segment. The segment relocatability type may also be specified. Each DSEG directive must specify the same segment relocatability type.

Format

```
    [label]    DSEG [, type-spec]
```

where *type-spec* is either PAGE or INPAGE. If the segment relocatability type is not specified, the relocatability type will be BYTE.

See Also

.SEG directive (Section 4.12),
ASEG directive (Section 4.15),
CSEG directive (Section 4.16)

4.20 DW

The DW directive is used to put word values into the object module. The expression may be absolute, relocatable or external, but all terms must have been defined when the expression is evaluated in the second pass. The low byte of the word value will go at a lower address than the high byte of the value.

Format

```
        [label]    DW    expr[,expr ...]
```

If more than one expression appears in a DW directive they must be separated by commas.

Example

An 80/PL program needs to use interrupt 3. An interrupt vector must be set up at memory location 18H (3 *8).

```
    .
EXTRN  INTERRUPTPROCEDURE
ASEG
ORG      18H
DB        (JMP      INTERRUPTPROCEDURE)
DW        INTERRUPTPROCEDURE
    .
```

(a) The procedure is declared EXTRN so that it becomes defined.

(b) Set the location counter to 18H in the absolute segment, since that is where the interrupt vector is to go.

(c) Use the DB directive to put the value of the JMP opcode (The operand of the JMP in the DB directive is needed for correct evaluation of the expression, even though the value of INTERRUPTPROCEDURE has no effect on the result).

(d) Put the address of the interrupt routine using DW (the value of a label or procedure is its address). The low byte of the address is at 19H, the high byte at 1AH.

Notes

1. The current location counter is incremented after processing each member of the list of expressions. This means that the value of the "$" term changes from expression to expression when there is a list of expressions.

See Also

DB directive (Section 4.17)

4.21 ELSE

The ELSE directive closes the last IF or ELSEIF conditional assembly block, and begins a new conditional assembly block. If the IF block and every ELSEIF block were ignored, then the statements within the ELSE block are assembled; otherwise they are ignored.

If the "cond" listing switch is set to *no*, the ELSE statement is not listed.

Format

```
        [label]    ELSE
```

Notes

1. The optional label "belongs" to the preceding IF or ELSEIF block. The label is defined if and only if the previous block was assembled.
2. The ELSE directive does not normally take an operand, but if the next token after the ELSE is IF, it is recognized as the ELSEIF directive.

See Also

IF directive (Section 4.27),
ELSEIF directive (Section 4.22),
ENDIF directive (Section 4.24)

4.22 ELSEIF

The ELSEIF directive closes the last IF or ELSEIF conditional assembly block, and begins a new conditional assembly block. This directive requires an expression as its operand. If the preceding if-block and any preceding elseif-blocks were ignored, and the expression is logically true (low bit is 1), the code following the ELSEIF directive and continuing to the matching ELSE, ELSEIF or ENDIF is assembled; otherwise, the lines are ignored. The expression must be defined when it is first encountered by the assembler – in particular it may contain no forward references.

If the "cond" listing switch is set to no, the ELSEIF statement is not listed.

Format

```
[label]    ELSEIF    expression
```

Notes

1. The optional label "belongs" to the preceding IF or ELSEIF block. The label is defined if and only if the previous block was assembled.
2. The directive ELSE, followed by blanks and then the token IF, is recognized as a variant of ELSEIF.

See Also

IF directive (Section 4.27),
ELSE directive (Section 4.21),
ENDIF directive (Section 4.24)

4.23 END

The last statement of every 80/AS program must be the END directive. If the program is a main program, the start address must appear in the operand field of the statement. If the operand field is blank, the module will be a subprogram with no start address.

Format

```
        [label]    END    [start-expr]
```

Example

A main program is to start at the label "BEGIN":

```
                .
                .
            END        BEGIN
```

4.24 ENDIF

The ENDIF directive terminates the current conditional assembly block.

Format

```
        [label]    ENDIF
```

Notes

1. The optional label "belongs" to the preceding IF, ELSEIF or ELSE block. The label is defined if and only if the previous block was assembled.

See Also

IF directive (Section 4.27),
ELSEIF directive (Section 4.22),
ELSE directive (Section 4.21)

4.25 EQU

The EQU directive defines a symbol and assigns it a 16-bit value. Symbols defined with EQU may not be redefined later in the program, and they must not have been previously defined (compare with SET, Section 4.31).

Because of the two-pass structure of 80/AS, limited forward referencing is possible; that is, the expression may contain symbols whose definition appears later in the program. When a symbol assignment is encountered in the first pass of the assembler, the expression is evaluated, and if each term is defined, the result is placed in the dictionary entry for the symbol. If, in the second pass, the expression still contains undefined terms, an error message is given.

Names may be EQUated to predefined register names.

Format

```
name    EQU    expression
```

Examples

The code fragments below illustrate the forward referencing capabilities and limitations. The first fragment,

```
X        EQU      Y        ; defined in second pass (=1)
Y        EQU      1        ; defined in first pass
```

is correct, while

```
X        EQU      Y        ; not defined at end of pass 2
Y        EQU      Z        ; defined in second pass (=1)
Z        EQU      1        ; defined in first pass
```

leaves X undefined.

Other possible EQU statements:

```
ACCUM    EQU      A        ; provide another name for reg A
         .
BUFFER:  DS       128
BUFSIZ   EQU      $-BUFFER ; length of buffer in bytes
```

See Also

SET directive (Section 4.31)

4.26 EXTRN

Sometimes it is necessary to reference data or labels within another module. The assembler must be instructed that certain references will not be defined in the current module, but appear elsewhere. This is done using the EXTRN directive. EXTRN expects a comma-separated list of symbols as its operand. These symbols are installed in the dictionary and defined, so that references to them will not cause errors. Any attempt to redefine an external symbol is an error.

Format

```
       [label]    EXTRN    name1[, name2 ... ]
```

See Also

PUBLIC directive (Section 4.30)

4.27 IF

The IF directive begins a conditional assembly block. This directive requires an expression as its operand; If the expression is logically true (low bit is 1), the code following the IF directive and continuing to the matching ELSE, ELSEIF or ENDIF is assembled; otherwise, the lines are ignored. The expression must be defined when it is first encountered by the assembler – in particular it may contain no forward references.

An IF directive may appear within another conditional assembly block; The conditional assembly mechanism can recurse up to 8 levels deep. Of course, if the outer conditional block is not being assembled, all inner blocks will be ignored. The lines will only be scanned to find the matching ENDIF.

If the "cond" listing switch is set to *no*, the IF statement is not listed.

Format

```
    [label]    IF    expression
```

See Also

ELSEIF directive (Section 4.22),
ELSE directive (Section 4.21),
ENDIF directive [Section 4.24]

4.28 NAME

The NAME directive allows the programmer to give the object module a name. If no module name is specified by the programmer, the object module is given the default name of "MODULE".

Format

```
        [label]    NAME    module-name
```

Example

```
                  .
               NAME      TEST@44
                  .
```

makes "TEST@44" the module name.

Notes

1. 80/LINK will complain if more than on of its input object modules has the same name, so the NAME directive should be used whenever multiple modules assembled by 80/AS are to be linked together.

4.29 ORG

The ORG directive is used to change the current location counter. The operand field of the ORG contains an expression whose value becomes the new value of the location counter. The expression must be either absolute or relocatable within the current segment, and must be completely defined when it is first encountered in pass one.

Format

```
     [label]    ORG    loc-expression
```

Examples

```
                .
             ORG    $+100H ; current loc ctr + 256
                .
```

would increment the current location counter by 256, and

```
                .
             ORG    3680H
                .
```

sets the location counter to 3680H, regardless of its previous value.

Notes

1. If the statement is labelled, the symbol in the label field is assigned the current address *before* changing the location counter.
2. The assembler does not check for multiple initialization of memory, so that setting the location counter to a lower address may cause errors that the assembler cannot detect.

See Also

DS directive (Section 4.18)

4.30 PUBLIC

When other modules will reference symbols defined in the current module, the linker must know the value of the symbol so the references can be resolved. The PUBLIC directive tells the assembler to put the name and value (at the end of pass 2) of each symbol in its comma-separated operand list into the object module. The appearance of a symbol in the PUBLIC statement does not constitute a definition; the symbol must be defined with a SET or EQU directive or by its use as a label.

Macros and register symbols may not be made public.

Format

```
    [label]    PUBLIC    name1[, name2 ... ]
```

See Also

EXTRN directive (Section 4.26)

4.31 SET

The SET directive defines a symbol and assigns it a 16-bit value. Symbols defined with SET may be redefined later in the program, using another SET directive (compare with EQU, Section 4.25).

Because of the two-pass structure of 80/AS, limited forward referencing is possible; that is, the expression may contain symbols whose definition appears later in the program. When a symbol assignment is encountered in the first pass of the assembler, the expression is evaluated, and if each term is defined, the result is placed in the dictionary entry for the symbol. If, in the second pass, the expression still contains undefined terms, an error message is given.

Names may be SET to predefined register names.

Format

```
        name    SET    expression
```

See Also

EQU directive (Section 4.25)

4.32 STKLN

The STKLN directive sets the size of the stack segment in the object module. If a program uses CALL or PUSH instructions, it will need to use the stack. One way to tell the linker how much space the program needs on the stack is the STKLN directive. Its operand, which must be an expression that evaluates to an absolute number, is the number of bytes that will be allocated in the stack segment for this module.

Format

```
        [label]    STKLN    expression
```

Notes

1. If many modules are to be linked together, only one of them need specify a stack length.
2. It is also possible to instruct 80/LOC to allocate a stack segment of the required size. In this case no STKLN statement is necessary.

5. Macros

A macro (in the 80/AS language) is a piece of text that is stored in the assembler's dictionary. The appearance of the name of the macro in the operation field of a statement (the "macro invocation" or "macro call") instructs the assembler to insert the entire text of the macro in place of the macro invocation statement. When the macro is defined it is possible to specify certain symbols that will be replaced by arguments that are given when the macro is called. This allows macros to act like subroutines, except that the code is expanded inline, resulting in faster execution at the expense of larger code size.

Macro definition and expansion takes place before the assembler performs its normal processing of the input line, so that the assembler proper is presented with a modified version of the source program. This has several implications for the programmer using the macro features of 80/AS:

- The macro processor "knows" only enough about the 80/AS language to allow it to recognize keywords and directives relevant to macros.
- The contents of the macro are ignored by the assembler during definition. Control lines such as ".INCLUDE" directives will not be processed when the macro is defined
- If a macro contains another macro definition, the inner macro will not become defined until the enclosing macro is expanded. (Even though the macro processor defines macros, the assembler must see the definition line in order to tell the macro processor to begin a definition.)

The directives recognized by the macro processor are:

MACRO	begins a macro definition (Section 5.1)
ENDM	ends a macro definition (Section 5.2)
REPT	begins a repeat block (Section 5.5)
IRP	begins an indefinite repeat block (Section 5.6)
IRPC	begins an indefinite repeat character block (Section 5.7)
LOCAL	instructs the macro processor to create unique names for symbols in the operand field so that repeated invocations of the macro do not cause multiple definition errors (Section 5.4)

Each of these directives is discussed individually below.

5.1 THE MACRO DIRECTIVE

A macro definition begins with the MACRO directive. This statement gives the macro its name, and tells which symbols should be replaced by actual arguments when the macro is expanded.

```
name     MACRO    [arg1[, arg2, ... ]]
```

5.1.1 The Macro Name

The label field of the statement must contain a name, which becomes the name of the macro. This name is used later to invoke the macro and cause its expansion. If the name already represents a macro, the old definition is deleted before the new definition is installed. It is an error to attempt to redefine a non-macro as a macro.

The statement

```
MAC1     MACRO
```

defines a macro with the name MAC1, whose definition begins with the following statement.

5.1.2 Formal Macro Parameters

The operand field of the statement contains a comma-separated list of symbols that are the formal parameters of the macro. During macro definition, if a symbol matching any of the formal parameters is encountered, its position in the stored code skeleton is marked for replacement by an argument of the macro when expanded.

The statement

```
MAC2     MACRO     XX, YY
```

defines the macro MAC2. Each occurrence of the tokens XX or YY within the definition will be replaced with a text string that is supplied when the macro is invoked for expansion.

Symbols that have special significance to the assembler, such as register names and instruction mnemonics, may be used as formal parameters, but since they will be replaced by actual arguments when the macro is expanded, they cannot be used in their normal way. For instance, if A is a formal parameter, the statement

```
MVI     A,10
```

will not cause the value 10 to be loaded into the accumulator (unless, of course, the argument substituted for A is itself A).

5.1.2.1 Forcing Recognition of Macro Parameters

Normally, formal parameters within the macro body will not be recognized unless they have the appearance of tokens; they must generally be separated from adjacent tokens by commas, non-alphanumeric operators or white space. Sometimes it is desirable to force recognition of a formal parameter when it is directly adjacent to another token, so that when the parameter is expanded it will make up only part of the resulting token. The character '&' is treated by the macro processor as a special kind of delimiter. If it immediately precedes or follows a formal parameter, the parameter is recognized, and the '&' is removed upon expansion.

For example,

```
MAC1    MACRO XX, YY
        DB      XXYY
        DB      XX&YY
        ENDM
        .
        MAC1    1, 2
```

will expand into

```
        DB      XXYY
        DB      12
```

Formal parameters within quoted strings will not be recognized unless adjacent to an '&'. Likewise, any '&' within a quoted string will not be removed unless directly adjacent to a formal parameter. If X is a formal parameter, the X in 'one X two' will not be recognized, but the X in 'three &X four' will.

When two or more ampersands are adjacent to a formal parameter, or separate two formal parameters, only one is removed when the macro is defined.

5.2 THE ENDM DIRECTIVE

The ENDM directive terminates the matching MACRO, REPT, IRP, or IRPC block. For the REPT, IRP, and IRPC cases, it also causes expansion of the block.

```
        ENDM
```

The ENDM directive may not be labelled, since the line containing the directive is not a part of the macro body. If a label is desired at the end of the macro, put it on a line by itself, immediately preceding the ENDM statement.

5.3 MACRO EXPANSION

A macro that has been previously defined is expanded when its name appears in the operation field of an instruction. Any arguments follow in the operand field, separated by commas.

```
[label]   name   [arg1[, arg2, ...]]
```

Each instance of the first formal parameter will be replaced with arg1, etc. If there are more arguments than formal parameters, they will be ignored. Any formal parameters that have no corresponding arguments will be set to null. A null argument is invisible to the assembler and takes up no space; it will not be a token separator.

5.3.1 Macro Arguments

The simplest form of argument is a string of characters which have no special meaning to the macro processor. The string is substituted directly for each occurrence of the corresponding formal parameter.

It is often necessary to include within an argument characters, such as blanks or commas, that the macro processor would recognize as token delimiters. Therefore, the macro processor has two different kinds of escape mechanism.

A single character may be escaped by preceding it with an '!'. Any character (including another '!') except a newline may be so escaped. The '!' is discarded when the argument is expanded, except when it is being used in an argument to a nested macro call, in which case it is kept.

Alternatively, an entire string of characters may be protected from the macro processor by enclosing them in angle brackets '<' and '>'. If the argument begins with an '<', the rest of the argument is scanned for the matching '>', and all the text between the angle brackets is treated as a single argument. Note that the outermost pair of brackets are removed, so that if the argument is to be used as the argument to another macro call, two sets of enclosing brackets will be needed.

A null macro argument that is followed by non-null arguments must sometimes be specified. This may be done either by using a quoted string that is empty ('') as the argument, or by omitting the argument entirely (the preceding and following commas are adjacent).

As an example of macro argument interpretation, assume that the macro BIGMAC was defined with the five parameters A, B, C, D, and E. Then the invocation

```
    BIGMAC  ,cheese, '', < tomato, mustard>, no! onions!!
```

would result in the following argument substitutions:

```
A  =  []                   (note the leading comma)
B  =  [cheese]
C  =  []                   (empty string)
D  =  [ tomato, mustard]   (leading blank, comma protected)
E  =  [no onions!]         (escaped space and !)
```

Occasionally it is necessary to pass the value represented by a text string (defined by SET or EQU) rather than the string itself. This is done by prefixing the argument with the '%' immediate evaluation character.

For example,

```
MAC1     MACRO    XX, YY
DATE     SET      1984
         DW       XX
         DW       YY
         ENDM
         .
DATE     SET      2001
         MAC1     DATE, %DATE
```

will expand into

```
DATE     SET      1984
         DW       DATE     (argument passed is DATE)
         DW       2001     (argument passed is 2001)
```

5.4 THE LOCAL DIRECTIVE

The LOCAL directive is used when a macro is being defined to specify one or more symbols that are to be replaced with generated names upon expansion. Each time the macro is expanded, a different generated name is used.

```
LOCAL    name1[, name2, ... ]
```

The LOCAL directive(s) must immediately follow the statement beginning the macro definition block (MACRO, IRP, IRPC, or REPT directive). More than one LOCAL directive may be used, as long as they are all at the beginning of the macro. The LOCAL directive may not be labelled.

Any time label definitions or EQU directives appear within a macro that will be expanded more than once, the LOCAL directive will need to be used. Otherwise, all expansions after the first will cause multiple definition of symbols.

The assembler generates replacement names of the form "??*nnnn*", where *nnnn* is 0000, 0001, 0002, etc. The programmer should therefore avoid using symbols of that form.

5.5 THE REPT DIRECTIVE

The REPT directive begins a macro definition that is also an implicit request for expansion. The operand field of the REPT statement contains a number or absolute expression that indicates the number of times that the macro should be expanded. Since the expansion call is implicit in the definition, no name is needed, and parameters are not allowed with the REPT directive.

```
[label]    REPT    expr
```

For example, the REPT block

```
        REPT    3
        RAR
        ENDM
```

would generate the following instructions:

```
        RAR
        RAR
        RAR
```

If the value in the operand field is 0 or no operand is present, no text is generated from the REPT block.

5.6 THE IRP DIRECTIVE

The IRP directive, like the REPT directive, begins a definition that is also an expansion.

```
[label]    IRP    param, <list>
```

Each occurrence of *param* within the body of the macro is replaced by a member of the argument list; one expansion of the macro is generated for each member. The (possibly empty) list is composed of a comma-separated list of text strings enclosed within angle brackets, which are treated exactly like arguments to a macro call (see Section 5.3.1). If the list is empty, a single expansion is generated, with the formal parameter replaced by a null argument.

If the source statements are:

```
        IRP     XX,<'declare', 'some', 'strings'>
        DB      XX
        ENDM
```

the macro processor would generate

```
        DB      'declare'
        DB      'some'
        DB      'strings'
```

Note that the quotes are preserved upon expansion, as long as the string is not empty.

5.7 THE IRPC DIRECTIVE

The IRPC directive also begins a definition that is an implicit call for expansion.

```
[label]    IRPC    param [,text]
```

The text argument is a sequence of characters, and if enclosed in angle brackets may contain delimiters such as blanks. The dummy parameter *param* is replaced by a single character from text, each character causing another expansion of the IRPC block. If the list is empty, a single expansion is generated, with the formal parameter replaced by a null argument. If the substitution string begins with %, the entire string is evaluated and the decimal representation of its value is used, one character at a time, to repeatedly expand the body of the IRPC block.

For example

```
IRPC    YY, 123
DB      YY
ENDM
```

would expand into

```
DB      1
DB      2
DB      3
```

5.8 THE EXITM DIRECTIVE

The EXITM directive causes the MACRO, REPT, IRP or IRPC block currently being expanded to be exited. It is typically used after an IF conditional assembly directive to terminate expansion in an error case.

For example

```
IF      COUNT GT 100H
.PRINT  1,'Won''t fit on one page'
EXITM
ENDIF
```

would cause termination of the current expansion if COUNT was greater than 256. An error code with severity 1 would also be issued – see Section 4.9 for a more complete discussion of the .PRINT directive.

5.9 MACRO EXAMPLES

This section presents two longer examples of the use of macros in the 80/AS assembler.

5.9.1 Generated Symbols Example

The following example shows the use of the LOCAL directive to create non-conflicting labels. The macro is a simple timing loop, which may be invoked many times. If the loop label was not made local to the expansion, all expansions after the first would cause multiple definition errors.

```
loc  obj           seq   text

              -     1 wait     macro    howlong
              -     2          local    loopxx
              -     3          mvi      a,howlong
              -     4 loopxx:  dcr      a
              -     5          jnz      loopxx
              -     6          endm
                    7          wait     250
0000 3efa       +   8          mvi      a,250
0002 3d         +   9 ??0001:  dcr      a
0003 c20200     +  10          jnz      ??0001
                   11          wait     75
0006 3e4b       +  12          mvi      a,75
```

```
0008 3d          +  13 ??0002: dcr     a
0009 c20800      +  14         jnz     ??0002
                    15         end
```

5.9.2 Argument Quoting Example

This example illustrates macro argument quoting mechanisms. Two macros, *quote* and *quote2*, are defined and then invoked with various quoted and escaped arguments. *Quote* invokes *quote2* and passes its arguments to the nested expansion. Compare sequence line 13 – the original invocation – with sequence line 17, which is the nested macro invocation. Things to note are:

1. The ampersands adjacent to the arguments within quoted strings are necessary to force recognition of the arguments. No substitution takes place for 'arg1' (line 11).
2. One set of angle brackets '<. . .>' is stripped off for each level of evaluation (compare arg1 evaluation).
3. The escape character '!' is not discarded when the argument being substituted is itself an argument to a nested macro call (compare arg2 and arg3 evaluation).

```
loc  obj            seq     text

                 -     1 quote   macro    arg1, arg2, arg3
                 -     2         db       '&arg1'
                 -     3         db       '&arg2'
                 -     4         db       '&arg3'
                 -     5         quote2   arg1, arg2, arg3
                 -     6         endm
                 -     7 quote2  macro    arg1, arg2, arg3
                 -     8         db       '&arg1'
                 -     9         db       '&arg2'
                 -    10         db       '&arg3'
                 -    11         db       'arg1'
                 -    12         endm
                      13         quote    <<1 2>>, !<1! 2!>, 1! 2!!
0000 3c312032      +  14         db       '<1 2>'
0004 3e            +
0005 3c312032      +  15         db       '<1 2>'
0009 3e            +
000a 31203221      +  16         db       '1 2!'
                   +  17         quote2   <1 2>, !<1! 2!>, 1! 2!!
000e 312032        +  18         db       '1 2'
0011 3c312032      +  19         db       '<1 2>'
0015 3e            +
0016 31203221      +  20         db       '1 2!'
001a 61726731      +  21         db       'arg1'
                      22         end
```

A. Error Messages

Following is a list of the error messages that may be produced, grouped by severity. For a discussion of return codes, see Section 2.2 and Section 2.4.

A.1 WARNINGS

Warnings are generated by 80/AS when a potential error has been encountered, even though an unambiguous and probably correct choice of actions is made.

- CAN'T LABEL THIS STATEMENT – A MACRO, EQU, or SET directive has been labelled. The label is assigned the current location and segment.
- END FOUND WHILE SKIPPING – The END directive was encountered while skipping lines from a false branch of a conditional assembly construct. All conditional blocks are closed and the END statement is processed.

A.2 ERRORS

Errors are generated by 80/AS when a statement contains one or more errors that are serious enough that the assembler cannot continue processing the statement. If an error of this type is encountered within a statement that appears to be an instruction (e.g. an undefined expression as the target of a CALL opcode), three bytes of NOP will be generated, allowing the programmer to patch the program at execution time.

- BAD BRANCH TARGET (DIFFERENT SEG) – The target of a short relative jump (Z80 instructions JR, JRC,JRNC, JRZ, and JRNZ) lies in another segment.
- BAD BRANCH TARGET (TOO FAR) – The target of a short relative jump (Z80 instructions JR, JRC,JRNC, JRZ, and JRNZ) is too far from the instruction (range -128 to +127 bytes).
- CAN'T CHANGE RELOCATABILITY TYPE – A .SEG, CSEG, or DSEG directive was seen that specified a relocatability (blank (BYTE), PAGE, or INPAGE) different from the relocatability already assigned to that segment.
- CAN'T ESCAPE A NEWLINE – A newline character (ASCII 10) followed the macro escape character "!". A blank was inserted after the escape character.
- CAN'T LOCATE INCLUDE FILE: ⟨file⟩ – The specified include file could not be found (or was not readable).
- COMMA EXPECTED – A comma was expected but not found at some point in the statement.

- CONSTANT OVERFLOW – When collecting a constant, its absolute value became greater than 65,535. Its value is the first N digits such that the first N+1 digits cause overflow.
- CONTROL STACK OVERFLOW – Too many .SAVE directives were used without intervening .RESTOREs. The directive is ignored.
- CONTROL STACK UNDERFLOW – More .RESTORE directives than .SAVE directives were encountered. The directive is ignored.
- DS EXPRESSION MUST BE ABSOLUTE – The expression in a DS directive was not an absolute value. The absolute part of the expression value is used.
- END OF LINE EXPECTED – After processing the statement, there was still something (other than a comment) left to scan.
- ENDM SYNTAX – The ENDM directive was not the first thing on the line (ENDM cannot be labelled).
- EXPRESSION MUST EVALUATE TO A BYTE – An expression whose value was outside the range -256 (0FF00H) to +255 (0FFH) was encountered in a context where a byte value was required. The high byte of the value is set to 0, giving a result in the range 0 to +255.
- EXPRESSION SYNTAX, OPERAND EXPECTED – An ill-formed expression was encountered. The value of the missing term is set to absolute 0.
- EXTERNAL NAME EXPECTED – Either no name followed an EXTRN directive, or the name was already defined.
- ILLEGAL CHARACTER – An ASCII control character (value < 32) other than a tab, or a printable ASCII character not recognized by 80/AS was found while scanning the input line. The character is discarded.
- ILLEGAL CHARACTER IN QUOTED STRING – An ASCII control character (value < 32) other than a tab was found in a quoted string. The character is discarded.
- ILLEGAL PUBLIC DECLARATION – A symbol name did not follow the PUBLIC directive.
- ILLEGAL PUBLIC REGISTER – Either a register name appeared in a PUBLIC directive, or a public name appeared as the target of a SET or EQU to a register name. The public attribute is removed, and the register type is retained.
- ILLEGAL REGISTER – A register or register pair was used as the operand of an instruction that could not reference that register or register pair (e.g. PUSH C).
- ILLEGAL USE OF A REGISTER – A register name appeared as an operand in an expression. The value of the term is set to absolute 0.
- INCLUDE FILE NAME OVERFLOW – An internal buffer overflowed when constructing an include file name from the string specified in an .INCLUDE directive and a default or user-supplied directory path. The directive is ignored.
- INCLUDE FILE STRING EXPECTED – A quoted string did not follow the .INCLUDE directive. The directive is ignored.
- INVALID BASE SPECIFIED FOR CONSTANT – The terminal character of a numeric token was not a valid base specifier (one of B, D, H, O, Q). The value of the number is set to 0.

- INVALID EXPRESSION IN ORG STATEMENT – The expression following the ORG directive has a relocatability that is not the same as the current segment. The expression relocatability is assumed to be that of the current segment.
- INVALID NUMERIC CONSTANT – BAD DIGIT – A character that is not a digit was found while scanning a number. The digits preceding the invalid character are used to determine the number's value.
- INVALID NUMERIC CONSTANT – DIGIT TOO LARGE – A digit that is not in the set of allowable digits for the given base was found while scanning a number. The digits preceding the invalid digit are used to determine the number's value.
- INVALID RELOCATABILITY TYPE – The segment relocatability type in a .SEG, CSEG, or DSEG directive is not one of PAGE, INPAGE, or blank (BYTE). The relocatability type is assumed to be BYTE.
- IRP REQUIRES A BRACKETED ARGUMENT LIST – An IRP macro directive did not have a bracketed argument list following the formal parameter and comma separator. The list is assumed to be null.
- LABEL ALREADY DEFINED – A label was encountered that had been previously defined. The symbol retains its initial value and relocatability.
- MISPLACED ELSE DIRECTIVE – An ELSE directive appeared when not within any conditional assembly block, or after another ELSE directive.
- MISPLACED ELSEIF DIRECTIVE – An ELSEIF directive appeared when not within any conditional assembly block, or after an ELSE directive.
- MISPLACED ENDIF DIRECTIVE – An ENDIF directive appeared when not within any conditional assembly block.
- MISPLACED ENDM DIRECTIVE – An ENDM directive appeared when not defining any macro.
- MISPLACED EXITM DIRECTIVE – An EXITM directive appeared when not expanding any macro.
- MISPLACED LOCAL DIRECTIVE – A LOCAL directive appeared that did not immediately follow a MACRO, IRP, REPT, or IRPC directive.
- MISSING END DIRECTIVE – The end of the primary source file was encountered before the END directive was seen. A generated END directive is appended to the input file.
- MULTIPLE NAME DIRECTIVES – More than one NAME directive was found. The module name is specified by the first NAME directive encountered.
- NO 8080 TRANSLATIONS FOR DADC OR DSBB – A DADC or DSBB Z80 instruction was encountered with the -z (map Z80 to 8080) option in effect. There are no single-instruction 8080 equivalents for the word add and subtract Z80 ops.
- NO MODULE NAME FOLLOWING NAME DIRECTIVE – The module will have the default name "MODULE".
- NO NAME WITH EQU DIRECTIVE – An EQU directive had no name in the label field.
- NO NAME WITH MACRO DIRECTIVE – A MACRO directive had no name in the label field. The definition is processed and then discarded.
- NO NAME WITH SET DIRECTIVE – A SET directive had no name in the label field.

- PRINT STRING EXPECTED – No string was found following a .PRINT directive.
- PUBLIC SYMBOL IS NOT DEFINED – A symbol specified in a PUBLIC directive did not appear in any definition context. The symbol will not appear in a PUBLIC record in the object output.
- PUBLICS NOT ALLOWED WITHIN COMMON – A symbol that has the public attribute was defined to have a relocatability of one of the common segments, or a symbol whose relocatability is that of one of the common segments appeared in a PUBLIC directive. The PUBLIC attribute is removed.
- REGISTER OPERAND REQUIRED – No register name was found where one was expected as an instruction operand.
- RELOCATION ERROR IN EXPRESSION – An expression contains one or more terms that have relocatabilities incompatible with each other or with the operator. The term in error is given the value of absolute 0.
- RIGHT PARENTHESIS EXPECTED – A parenthesized expression is missing one or more right parentheses.
- RST VALUE MUST BE BETWEEN 0 AND 7 – The RST instruction was either not followed by a number, or the value was outside the range 0 - 7.
- SEGMENT ADDRESS WRAPAROUND – The location counter for a segment has gone past 65,535 (0FFFFH). The next address will be location 0.
- SEGMENT NAME EXPECTED – An invalid segment name was found following the .SEG directive.
- STKLN EXPRESSION MUST BE ABSOLUTE – The expression in a STKLN directive was not an absolute value. The absolute part of the expression value is used.
- STRING CAN'T REPRESENT A VALUE – A string more than two characters long was found where a number was required. The string is taken to have a value of 0.
- STRING TOO LONG – A string longer than 255 characters was found. The excess characters are truncated.
- SYMBOL ALREADY DEFINED – The target of an EQU was already defined, or the target of a SET was already defined other than with a SET directive. The previous definition remains in effect.
- SYMBOL ALREADY DEFINED AS A NON-MACRO – An attempt was made to define a macro whose name was already defined as a non-macro symbol. The macro definition is scanned and then discarded.
- SYMBOL EXPECTED – A symbol was expected but not found following a MACRO, LOCAL, IRP, or IRPC directive.
- TITLE OR SUBTITLE STRING EXPECTED – No string was found following a .TITLE or .SUBTITLE directive.
- TOKEN TOO LONG – A token (other than a quoted string) was found that was longer than 31 characters. The token is truncated to the first 31 characters.
- TOO MANY SEGMENTS – More than 249 named common segments were defined with SEG directives. Excess segment definition directives are ignored.
- UNBALANCED BRACKETS – A newline was found while processing a bracketed macro argument. Indicates one or more missing closing angle brackets (>).

- UNCLOSED STRING – The end of the input line was encountered while scanning a quoted string. The string is terminated as if there had been a closing quote.
- UNDEFINED EXPRESSION – An undefined expression was found where a defined expression was required. The expression is given the value of absolute 0.
- UNDEFINED EXPRESSION IN SYMBOL ASSIGNMENT – An undefined expression was found as the operand of a SET or EQU directive in pass 2. The symbol is given the value of absolute 0.
- UNKNOWN OPCODE – The operation field of a statement does not contain a recognizable directive, instruction mnemonic, or macro name.

A.3 SEVERE ERRORS

These errors are the most severe errors that the user should encounter in normal operation. Severe errors are errors from which the assembler cannot recover, and they cause immediate termination of the current assembly. These errors fall into two general classes. They may be caused by some dynamic or static space overflow within the assembler, and the solution is to reduce the size and/or complexity of the program. Or, they may indicate some problem with the environment within which 80/AS runs. I/O errors generally fall into this category.

Messages associated with system errors will generally have an associated reason message indicating the type of problem encountered (i.e. "File table overflow", "No space left on device").

- CAN'T GENERATE UNIQUE FILE NAME: ⟨pattern⟩ – 80/AS generates temporary file names using the process id and any one of the lowercase letters. None of these names could be used because all the files already existed.
- CREATING FILE MIGHT ERASE SOURCE FILE: ⟨file⟩ – The specified file was not created because its name matched the source file name.
- DICTIONARY OVERFLOW – The assembler's dictionary has overflowed. Too many symbols have been defined.
- DYNAMIC MEMORY OVERFLOW – There is not enough memory for the assembly. This can happen if there are a large number of macros defined, and/or many include files are open at the same time.
- EXPRESSION STACK OVERFLOW – Too many partial results have been pushed on the expression stack. Simplify the expression.
- IF STACK OVERFLOW – The conditional assembly stack has overflowed. Reduce nesting of conditional assembly blocks.
- INPUT STACK OVERFLOW – The macro processor's input stack has overflowed. There are too many nested macro expansions and open include files.
- I/O ERROR ON CLOSE: ⟨reason⟩ – A system I/O error occurred when closing a file.
- I/O ERROR ON OPEN: ⟨file⟩: ⟨reason⟩ – A system I/O error occurred when opening the specified file.
- I/O ERROR ON READ: ⟨reason⟩ – A system I/O error occurred when reading a file.

- I/O ERROR ON SEEK: ⟨reason⟩ – A system I/O error occurred when seeking in a file.
- I/O ERROR ON WRITE: ⟨reason⟩ – A system I/O error occurred when writing a file.
- MISSING NUMERIC ARGUMENT – An invocation option that required a numeric value to follow was specified without the value.
- NOTHING REMAINS AFTER STRIPPING THE SUFFIX: ⟨file⟩ – An input file had only an extension.
- SORT FAILED – The sort program returned a non-zero status.
- TOO MANY INCLUDE DIRECTORIES – The list of include directories given with the -I invocation option has overflowed an internal buffer. Reduce the number or length of the include directories specified.
- TOO MANY MACRO PARAMETERS – When defining a macro the macro parameter stack overflowed. Reduce the number of formal parameters in the definition.
- TOO MANY OPEN FILES – The I/O processor's file table filled up. Reduce the number of nested include files.

A.4 FATAL ERRORS

Fatal errors indicate an internal 80/AS failure. They should never be encountered by the user.

- ASSEMBLER FAILURE – ⟨message⟩ – All internal errors are of this form. The associated message indicates the particular mode of failure.

B. Glossary of Directives

Following is a list of 80/AS assembler directives and their formats. For a more complete discussion of each, along with some examples, see the chapter on directives, Chapter 4 and the chapter on macros, Chapter 5.

`[label]` `.COND`

List lines that were ignored because of a conditional assembly directive.

`[label]` `.EJECT`

Generate an eject in the listing.

`[label]` `.GEN`

List macro expansions.

`[label]` `.INCLUDE 'file-name'`

Include the file *file-name*.

`[label]` `.LIST`

Enable listing.

`[label]` `.NOCOND`

Do not list lines that were ignored because of a conditional assembly directive.

`[label]` `.NOGEN`

Do not list macro expansions.

`[label]` `.NOLIST`

Disable all listing.

`[label]` `.PRINT expression, 'message'`

Signal an error of severity *expression*, and add *message* to the error log and listing.

`[label]` `.RESTORE`

Restore saved listing controls.

`[label]` `.SAVE`

Save listing controls.

[label] .SEG seg-name [, type-spec]

Set the current segment to the STACK, MEMORY, or common segment *seg-name*. A *seg-name* of // indicates the unnamed common segment. The relocatability will be BYTE unless either PAGE or INPAGE is given as the *type-spec*.

[label] .SUBTITLE 'subtitle-string'

Set the current subtitle to *subtitle-string*.

[label] .TITLE 'title-string'

Set the current title to *title-string*.

[label] ASEG

Set the current segment to the absolute segment.

[label] CSEG [, type-spec]

Set the current segment to the code segment. The relocatability will be BYTE unless either PAGE or INPAGE is given as the *type-spec*.

[label] DB expr-or-string [, expr-or-string ...]

Initialize data with byte values and/or quoted strings.

[label] DS expr

Reserve expr bytes starting at the current location.

[label] DSEG [, type-spec]

Set the current segment to the data segment. The relocatability will be BYTE unless either PAGE or INPAGE is given as the *type-spec*.

[label] DW expr [, expr ...]

Initialize data with word values.

[label] ELSE

Begin else clause of conditional assembly block.

[label] ELSEIF expression

Begin elseif clause of conditional assembly block.

[label] END [start-expr]

Indicates end of source program. If *start-expr* is present, module is a main program with given start address.

[label] ENDIF

End conditional assembly block.

ENDM

End a macro, rept, irp, or irpc definition.

name EQU expression

Define symbol *name* to have value *expression*.

[label] EXITM

Terminate expansion of the current macro, rept, irp, or irpc block.

`[label]` EXTRN `name1[, name2 ... ]`

Define the symbols *name1*, ... to be external.

`[label]` IF `expression`

Begin if clause of conditional assembly block.

`[label]` IRP `param, <list>`

Define and expand an indefinite repeat block with formal parameter *param* and actual arguments supplied by the (possibly empty) comma-separated *list* elements.

`[label]` IRPC `param [,text]`

Define and expand an indefinite repeat (character) block with formal parameter *param* and actual arguments supplied by successive characters from *text*.

LOCAL `name1 [, ... ]`

Within a macro block, define the symbols *name1*, ... to be local symbols, causing the assembler to replace each instance of the name with a unique generated name.

MACRO `[param1, ...]`

Begin a macro definition whose name is *name*, with formal parameters *param1*, ...

`[label]` NAME `module-name`

Give the object module the name *module-name*.

`[label]` ORG `loc-expression`

Set the location counter to *loc-expression*.

`[label]` PUBLIC `name1[, name2 ... ]`

Give the symbols *name1*, ... the public attribute, causing PUBDEF records to be generated.

`[label]` REPT `count`

Define a repeat block with a repeat count of *count*.

`name` SET `expression`

Define symbol *name* to have value *expression*. The symbol may be redefined with another SET directive.

`[label]` STKLN `expression`

Set the size of the stack segment to *expression* bytes.

Index