Z80AS

Z80 Macro Assembler

for CP/M-80

Original ZSM4 manual written by Hector Peraza

Updated for Z80AS by Ladislau Szilagyi, Jan'2022

V4.7

1 Introduction

Z80AS is a macro-assembler for Z80 microcomputers, running on the CP/M operating system.

It runs on Z80-based computers, assembles Z80 source code and is compatible with the HiTech C compiler, ZAS assembler and LINK linker.

It is a major rewrite of the Z80ASM assembler which first appeared on CP/M US User Group, Vol 16.

The original source was Copyright (C) 1977, Lehman Consulting Services and was put into the public domain.

It was further modified by Ray Halls in 1982 and Neil Harrison in 1983. Last known version was 2.8.

It was further modified by Hector Peraza, starting from 2017.

Z80AS is actually an adaptation of the ZSM4 macro-assembler v4.6 published by Hector Peraza on GitHub (https://github.com/hperaza/ZSM4), made by Ladislau Szilagyi in December 2021 – January 2022.

Thanks to the high quality of the work done by Hector Peraza, the adaptation was easy, actually the most difficult part was writing a new OBJ format object code generator.

2 Motivation

The main target of the adaptation was to obtain an assembler compatible with ZAS, the assembler that is used by the HiTech C Compiler, but superior as performance.

The biggest ZAS problem is related to its size (38KB), and because of this ZAS is unable to assemble large source files (too small free space remains for the symbols). Z80AS's size is less than 22KB, that means 16KB more memory is available to store the symbols, compared to ZAS.

For example, ZAS fails to assemble the CP/M BDOS source. In contrast, Z80AS succeeds to assemble even a larger file, the CP/M BDOS and BIOS, concatenated.

Often, using HiTech's C compiler to compile large C files is not possible because ZAS fails to assemble the intermediate file produced by the compiler ("out of memory" error message).

Z80AS is compatible with HiTech’s ZAS and produces object files compatible with HiTech’s LINK linker.

Therefore, it can substitute ZAS, without disturbing the use of HiTech's C compiler.

Try this:

>PIP OLDZAS.COM=ZAS.COM (save a copy of ZAS.COM, just in case)
>PIP ZAS.COM=Z80AS.COM

Then, try to compile or build an executable file using the HiTech's C command:

>C -V -C anyfile.c (just compile anyfile.c)

Or

>C -V anyfile.c (build anyfile.COM)

Z80AS will be executed, instead of ZAS, building the requested object code.

Z80AS main features are:

- Assembles Z80 code using the standard Zilog/Mostek mnemonics.
- The undocumented Z80 instructions are supported.
- Supports conditional assembly. Conditionals may be nested up to 8 levels.
- Supports a complete standard macro facility including IRP, IRPC, REPEAT, local variables and EXITM. Nesting of macros is limited only by memory.
- Produces a .OBJ object code file suitable for linking with the HiTech LINK linker.

3 Running Z80AS

3.1 Invoking the assembler

The command to run Z80AS is

Z80AS *[-L[listfile]] [-Oobjfile] [-J] [-Q] [-X] [-D] sourcefile[.as]*

where *objfile*, *listfile* and *sourcefile* are file specifications for the output object file, output listing file and input source file respectively. The options -L, -O, -J, -X, -D and -Q are not case sensitive.

The file names must be valid CP/M file names and can specify also a device name; the source file extension default is “AS”.

The option -L specifies that a listing will be requested; if a valid file name follows, it designates the file that will contain the listing, otherwise the source file name will be used, with the .LST extension.

The option -O specifies the name of the object file, otherwise the source file name will be used, with the .OBJ extension.

The option -J requests the assembler to optimize all the “JP address” or “JP cond,address” (cond=Z,NZ,C,NC) statements that can be changed into “JR address” or “JR cond,address”.

When using the -J option, the JOPT pseudo operator can be used to select specific source code areas where the optimization will be applied.

The option -X will store to the object file only global symbols; by default, all symbols (local & global) are stored to the object code file.

The option -D requests the assembler to fill the DEFS buffers with zero, in the object code; by default, the DEFS buffers are not initialized.

The option -Q sets the assembler in “brief” mode; no messages will be typed, excepting the error messages.

In the normal (“verbose”) mode, the assembler types an initial message:

“Z80AS Macro-Assembler V4.7”

and the final messages:

“nnn jump optimizations done” (if -J option was used),
“Errors: nnn”
“Finished.”

The default extensions are as follows:

File	
Relocatable object file	OBJ
Listing file	LST
Source file	AS

The device names are as follows:

Device	
Disk drives	A:, B:, … J:
Line printer	LST:
Console	TTY: or CON:

Examples:

Z80AS -LTTY: TEST Z80AS	Assemble the source file TEST.AS, place the object file in TEST.OBJ and the listing on the console.
-L -OT.OBJ TEST.AS	Assemble TEST.AS, place the object file in T.OBJ and the listing on TEST.LST.
Z80AS X	Assemble X.AS, place the object file in X.OBJ

3.2 Source file format

Input source lines of up to 160 characters in length are acceptable. All symbols, opcodes and pseudo-opcodes can be typed in lower case or uppercase.

3.2.1 Statements

Source files input to Z80AS consist of statements of the form:

[*label*:[:]] [*operator*] [*arguments*] [;*comment*]

Statements do not need to begin in column 1. Multiple blanks or tabs may be used to improve readability.

If a label is present, it is the first item in the statement and must be immediately followed by a colon. If the label is followed by two colons, it is declared as GLOBAL. For example:

```
FOO::  RET
```
is equivalent to
```
GLOBAL  FOO
FOO:   RET
```

The next item after the label, or the first item on the line if no label is present, is an operator. An operator may be a Z80 mnemonic, pseudo-op, macro call or expression. The evaluation is as follows:

1. Macro call
2. Mnemonic/Pseudo operation
3. Expression

The arguments following the operator will, of course, vary in form according to the operator.

A comment always begins with a semicolon and ends with a carriage return. A comment may be a line by itself or it may be appended to a line that contains a statement.

3.2.2 Symbols

Z80AS symbols may contain up to 31 characters. Upper and lower case are distinct. The following characters are legal in a symbol: A-Z , 0-9 , $, ? , _ , @ . A symbol may not start with a digit, with the exception of ‘special’ symbols containing only up to 5 digits; these ‘special’ symbols must be referenced appending a “b” or “f” character (only lowercase). These ‘special’ symbols can be used in more than one location (as “temporary” labels):
Example:

```
        JR      1f
        …
1:      …
        JR      1f
        …
1:
```

If a symbol reference is followed by ## it is declared GLOBAL (external).

All undefined symbols will be considered GLOBAL external symbols.

3.2.3 Numeric Constants

The default base for numeric constants is decimal. This may be changed by the .RADIX pseudo-op. Any base from 2 (binary) to 16 (hexadecimal) may be selected. When the base is greater than 10, A-F are the digits following 9. If the first digit of a number is not numeric, the number must be preceded by a zero.

Numbers are 16-bit unsigned quantities. A number is always evaluated in the current radix unless one of the following special notations is used:

*nnnn*B	Binary
*nnnn*D	Decimal
*nnnn*O	Octal
*nnnn*Q	Octal
*nnnn*H	Hexadecimal

Overflow of a number beyond two bytes is ignored and the result is the low order 16-bits.

A character constant is a string comprised of zero, one or two ASCII characters, delimited by quotation marks, and used in a non-simple expression. For example, in the statement

```
DB      'A'+1
```

'A' is a character constant. But the statement

```
DB      'A'
```

uses 'A' as a string because it is in a simple expression. The rules for character constant delimiters are the same as for strings.

A character constant comprised of one character has as its value the ASCII value of that character. That is, the high order byte of the value is zero, and the low order byte is the ASCII value of the character. For example, the value of the constant 'A' is 41H.

A character constant comprised of two characters has as its value the ASCII value of the first character in the *low* order byte and the ASCII value of the second character in the *high* order byte. For example, the value of the character constant 'AB' is 41H+42H*256.

3.2.4 Strings

A string is comprised of zero or more characters delimited by quotation marks. Either single or double quotes may be used as string delimiters. The delimiter quotes may be used as characters if they appear twice for every character occurrence desired. For example, the statement

DB "I am ""great"" today" stores the string I am "great" today

If there are zero characters between the delimiters, the string is a null string.

3.3 Expression Evaluation

3.3.1 Arithmetic and Logical Operators

The following table list the allowed operators and their precedence:

(the operators are case sensitive)

Operator	Function	Precedence
NUL	Test for Null argument	1
LOW .low.	Take LOW byte	2
HIGH .high.	Take HIGH byte	2
*	Unsigned Multiply	3
/	Unsigned Divide	3
MOD .mod.	Unsigned Module	3
SHR .shr.	Shift Right	3
SHL .shl.	Shift Left	3
-	Unary Minus	4
+	Add	5
-	Subtract	5
EQ .eq. =	Equal	6
NE	Not Equal	6
LT .ult. <	Less Than	6
LE	Less Than or Equal	6
GT .ugt. >	Greater Than	6
GE	Greater Than or Equal	6
LESS .lt.	Signed Less Than	6
NOT .not.	Bitwise Not	7
AND .and.	Bitwise And	8
OR .or. ^	Bitwise Or	9
XOR .xor.	Bitwise Exclusive Or	9

Parentheses are used to change the order of precedence. During evaluation of an expression, as soon as a new operator is encountered that has precedence less than or equal to the last operator encountered, all operations up to the new operator are performed. That is, sub-expressions involving operators of higher precedence are computed first.

All operators except +, -, *, / must be separated from their operands by at least one space.

The byte isolation operators (HIGH, LOW) isolate the high or low order 8 bits of an Absolute 16-bit value. If a relocatable value is supplied as an operand, HIGH and LOW will treat it as if it were relative to location zero.

3.3.2 Modes

All symbols used as operands in expressions are in one of the following modes: Absolute, Data Relative, Uninitialized Data Relative, Program (Code) Relative.

Symbols assembled under PSECT text,abs (or ASEG) are in Absolute mode.

Symbols assembled under PSECT data (or DSEG) are in Data Relative mode.

Symbols assembled under PSECT bss are in Uninitialized Data Relative mode.

Symbols assembled under PSECT text (or CSEG) are in Code Relative mode.

In any operation other than addition or subtraction, the mode of both operands must be Absolute.

If the operation is addition, the following rules apply:

1. At least one of the operands must be Absolute.
2. Absolute + *mode* = *mode*

If the operation is subtraction, the following rules apply:

1. *mode* - Absolute = *mode*
2. *mode* - *mode* = Absolute

where the two *mode*s are the same.

Each intermediate step in the evaluation of an expression must conform to the above rules for modes, or an error will be generated. For example, if FOO, BAZ and ZAZ are three Program Relative symbols, the expression

FOO + (BAZ - ZAZ)

is legal because the first step (BAZ - ZAZ) generates an Absolute value that is then added to the Program Relative value, FOO.

3.3.3 Externals

Aside from its classification by mode, a symbol is either External or not External. An External symbol is a symbol declared GLOBAL but not defined in the source file. Also, all undefined symbols will be considered external. An External value must be assembled into a two-byte field (singe-byte Externals are not supported.) The following rules apply to the use of Externals in expressions:

1. Externals are legal only in addition and subtraction.
2. If an External symbol is used in an expression, the result of the expression is always External.
3. When the operation is addition, either operand (but not both) may be External.
4. When the operation is subtraction, only the first operand may be External.

3.4 Pseudo operators

3.4.1 ASEG

Syntax:

ASEG

ASEG sets the location counter to an absolute segment of memory. The location of the absolute counter will be that of the last ASEG (default is 0), unless an ORG is done after the ASEG to change the location.

ASEG is equivalent to PSECT text,abs.

3.4.2 CSEG

Syntax:

CSEG

CSEG sets the location counter to the code relative segment of memory. The location will be that of the last CSEG (default is 0), unless an ORG is done after the CSEG to change the location. CSEG is the default condition of the assembler. CSEG is equivalent to PSECT text.

3.4.3 DEFB, DEFM, DB (Define Byte)

Syntax:

DB *exp*[,*exp*...][,*string*...]
DB *string*[,*string*...][,*exp*...]
DEFB *exp*[,*exp*...][,*string*...]
DEFB *string*[,*string*...][,*exp*...]
DEFM *string*[,*string*...]

The arguments to DB are either expressions or strings. DB stores the values of the expressions or the characters of the strings in successive memory locations beginning with the current location.

Expressions must evaluate to one byte. (if the high byte of the result is 0 or 255, no error is given; otherwise, an A error results.)

Strings of three or more characters may not be used in expressions (i.e., they must be immediately followed by a comma or the end of the line). The characters in a string are stored in the order of appearance, each as a one-byte value with the high order bit set to zero.

3.4.4 DEFC, DC (Define Character)

Syntax:

DC *string*
DEFC *strin*

DC stores the characters in *string* in successive memory locations beginning with the current location counter. As with DB, characters are stored in order of appearance, each as a one-byte value with the high order bit set to zero. However, DC stores the last character of the string with the high order bit set to one. An error will result if the argument to DC is a null string.

3.4.5 DEFF (Define Floating point constant)

Syntax

DEFF fpconst[,fpconst…]

The *fpconst* must respect the floating point number's syntax: [+|-][digits][.][digits][e|E][+|-][digits]

Examples of valid floating point constants:

- 10
- 123.4567
- -12.56E5
- 56.876e-7

DEFF stores the value of one floating point constant in 4 successive memory locations beginning with the current location. The floating point number format is as follows: 1 bit sign, 7 bits exponent, 24 bits mantissa, normalized. It is stored with the mantissa in the low order bytes, and the sign bit is the most significant bit of the most significant byte.

If the floating point constant is too big or too small, the F (overflow) error will be reported in the listing.

3.4.6 DEFS, DS (Define Space)

Syntax:

DS *exp*
DEFS *exp*

DS reserves an area of memory. The value of *exp* gives the number of bytes to be allocated. All names used in *exp* must be previously defined (i.e., all names known at that point on pass 1). Otherwise, a V error is generated during pass 1 and a U error may be generated during pass 2.

The area of memory is initialized with zero only if the -D command line option was used; by default, the reserved area of memory is not initialized.

3.4.7 DEFW, DW (Define Word)

Syntax:

DW *exp*[,*exp*...]
DEFW *exp*[,*exp*...]

DW stores the values of the expressions in successive memory locations beginning with the current location counter. Expressions are evaluated as 2-byte (word) values.

3.4.8 DEFZ (Define Zero-terminated string)

Syntax:

DEFZ *string*

DEFZ stores the characters in *string* in successive memory locations beginning with the current location counter and adds an extra zero byte at the end of the string.

3.4.9 DSEG

Syntax:

DSEG

DSEG sets the location counter to the Data Relative segment of memory. The location of the data relative counter will be that of the last DSEG (default is 0), unless an ORG is done after the DSEG to change the location. DSEG is equivalent to PSECT data.

3.4.10 END

Syntax:

END [*exp*]

The END statement specifies the end of the program. If specified, the *exp* argument will be set as start address.

3.4.11 GLOBAL

Syntax:

GLOBAL *name*[,*name*...]

GLOBAL should be followed by one or more symbols (comma separated) which will be treated by the assembler as global symbols, either internal or external depending on whether they are defined within the current module or not.

3.4.12 EQU

Syntax:

name **EQU** *exp*

EQU assigns the value of *exp* to *name*. If *exp* is external, an error is generated. If *name* already has a value other than *exp*, an M error is generated.

3.4.13 INCLUDE

Syntax:

INCLUDE *filename*
or
***INCLUDE** *filename*

The INCLUDE pseudo-op assembles source statements from an alternate source file into the current source file. Use of INCLUDE eliminates the need to repeat an often-used sequence of statements in the current source file.

filename is any valid file specification, as determined by the operating system. Defaults for file name extensions and device names are the same as those in a Z80AS command line.

The INCLUDE file is opened and assembled into the current source file immediately following the INCLUDE statement. When end-of-file is reached, assembly resumes with the statement following INCLUDE. On the listing, a C character is printed between the assembled code and the source line on each line assembled from an INCLUDE file.

Nested INCLUDEs are allowed up to a level of 5.

The file specified in the operand field must exist. If the file is not found, a V error (value error) is generated, and the INCLUDE is ignored.

3.4.14 JOPT (Jump optimizations enable/disable)

Syntax:

JOPT ON or **JOPT OFF**

JOPT ON enables, JOPT OFF disables the jump optimizations requested by the -J command line option.

All the following statements will be processed according this setting, until another JOPT will be encountered or until the end of file.

This way, we can select the exact area where jump optimizations will be applied.

JOPT must be used in connection with the -J command line option, otherwise it has no effect.

3.4.14 ORG (Define Origin)

Syntax:

ORG *exp*

ORG changes the current segment to Absolute Code (ASEG, or PSECT text,abs), and sets a new program counter (PC).

The value of *exp* will be the new value of the location counter in the absolute code segment. All names used in *exp* must be known on pass 1 and the value of *exp* must be Absolute.

3.4.15 PAGE, EJECT, FORM

Syntax:

PAGE [*exp*]
EJECT (or ***EJECT)**
FORM

EJECT, FORM or PAGE causes the assembler to start a new output page. The value of *exp*, if included in the PAGE statement, becomes the new page size (measured in lines per page) and must be in the range 10 to 255.

The default page size is 60 lines per page.

The assembler puts a form feed character in the listing file at the end of a page.

3.4.16 PSECT

Syntax:

PSECT text[,abs]
PSECT data
PSECT bss
PSECT name (custom segments)

PSECT text is equivalent to CSEG.
PSECT text,abs is equivalent to ASEG.
PSECT data is equivalent to DSEG.
PSECT bss sets the location counter to the Uninitialized Data Relative segment of memory.

Up to 3 different ‘custom’ segments can be used (a segment name other than ‘text’, ‘data’ or ‘bss’; the ‘name’ used as parameter can be any valid symbol with up to 4 characters).

Note that the segment names (text,data,abs or the ‘custom segment names’) are case sensitive; in order to maintain compatibility with the HiTech C compiler, lowercase names must be used.

3.4.17 SUBTTL

Syntax:

SUBTTL *text*
or
***HEADING** *text*

SUBTTL specifies a subtitle to be listed on the line after the title on each page heading. *text* is truncated after 60 characters. Any number of SUBTTLs may be given in a program.

3.4.18 TITLE

Syntax:

TITLE *text*
or
***TITLE** *text*

TITLE specifies a title to be listed on the first line of each page. If more than one TITLE is given, a Q error results.

3.4.19 $PRINTX

Syntax:

$PRINTX *delimiter...text...delimiter*

The first non-blank character encountered after $PRINTX is the delimiter. The text that follows is listed on the terminal during assembly until another occurrence of the delimiter is encountered. $PRINTX is useful for displaying progress through a long assembly or for displaying the value of conditional assembly switches. For example:

```
IF     CPM
$PRINTX /CPM version/
ENDIF
```

NOTE

$PRINTX will output on both passes. If only one printout is desired, use the IF1 or IF2 pseudo-op. For example:

```
IF2
IF CPM
$PRINTX /CPM version/
ENDIF
ENDIF
```

will only print if CPM is true and Z80AS is in pass 2.

3.4.20 $RADIX

Syntax:

$RADIX *exp*

The default base (or radix) for all constants is decimal. The $RADIX statement allows the default radix to be changed to any base in the range of 2 to 16. For example:

```
LD      A,0FFH
$RADIX  16
LD      B,0FF
```

The two LDs in the example above are identical. The *exp* in a $RADIX statement is always in decimal radix, regardless of the current radix.

3.4.21 Conditional Pseudo Operations

The conditional pseudo operations are:

IF/**IFT** *exp*	True if *exp* is not 0.
COND exp	True if *exp* is not 0.
IFF *exp*	True if *exp* is 0.
IF1	True if pass 1.
IF2	True if pass 2.
IFDEF *symbol*	True if *symbol* is defined or has been declared External.
IFNDEF *symbol*	True if *symbol* is undefined or not declared External.
IFB <*arg***>**	True if *arg* is blank. The angle brackets around *arg* are required.
IFNB <*arg***>**	True if *arg* is not blank. Used for testing when dummy parameters are supplied. The angle brackets around *arg* are required.
IFIDN <*arg1***>,<***arg2***>**	True if the string *arg1* is IDeNtical to the string *arg2*. The angle brackets around *arg1* and *arg2* are required.
IFDIF <*arg1***>,<***arg2***>**	True if the string *arg1* is DIFferent from the string *arg2*. The angle brackets around *arg1* and *arg2* are required.

All conditionals use the following format:

IF*xx* [*argument*]
.
[**ELSE**
.]
ENDIF

Conditionals may be nested to a maximum of 10 levels. Any argument to a conditional must be known on pass 1 to avoid V errors and incorrect evaluation. For IF, IFT and IFF the expression must involve values which were previously defined and the expression must be absolute. If the name is defined after an IFDEF or IFNDEF, pass 1 considers the name to be undefined, but it will be defined on pass 2.

COND is equivalent to IF.

3.4.21.1 ELSE

Each conditional pseudo operation may optionally be used with the ELSE pseudo operation which allows alternate code to be generated when the opposite condition exists. Only one ELSE is permitted for a given IF, and an ELSE is always bound to the most recent, open IF. A conditional with more than one ELSE or an ELSE without a conditional will cause a C error.

3.4.21.2 ENDIF

Each IF must have a matching ENDIF to terminate the conditional. Otherwise, a T error is generated at the end of each pass. An ENDIF without a matching IF causes a C error.

ENDC is equivalent to ENDIF.

3.4.22 Listing Control Pseudo Operations

Output to the listing file can be controlled by the LIST pseudo-op:

LIST *option*,*option*,...

If a listing is not being made, the LIST pseudo-op has no effect. Multiple options can be specified in the same LIST statement, separated by commas. The options are the following:

OFF (or ***LIST OFF)** Suppresses the listing until a LIST ON command.

ON (or ***LIST ON)** Turns on the listing file after a LIST OFF statement. This is the default condition.

COND Turns on the listing of false conditional blocks. This is the default condition.

NOCOND Suppresses listing of false conditionals.

SYMBOL Generates a symbol table at the end of the listing. This is the default condition.

NOSYMBOL Suppress printing of the symbol table at the end of the listing file.

SORT Sort the symbol table generated with the SYMBOL option.

NOSORT Do not sort the symbol table. This typically results in shorter assembly times. This is the default condition.

MACROS List the complete macro text for all MACRO/REPT/IRP/IRPC expansions.

NOMACROS Suppress listing of all text and code produced by macros.

XMACROS List only the macro source lines that generate object code. This is the default condition.

The following list control pseudo-operators are also recognized:

$XLIST same as LIST OFF

$LIST same as LIST ON

$LALL same as LIST MACROS

$SALL same as LIST NOMACROS

$XALL same as LIST XMACROS

3.4.23 Relocation Pseudo Operations

Z80AS supports all four relocatable areas defined by the OBJ object code relocatable format: text abs (ASEG, Absolute), text (CSEG, Code), data (DSEG, Data) and uninitialized data bss.

The default mode for the assembler is text (Code Relative). That is, assembly begins with a CSEG automatically executed and the location counter in the Code Relative mode, pointing to location 0 in the Code Relative segment of memory. All subsequent instructions will be assembled into the Code Relative segment of memory until a ASEG (PSECT text,abs) or DSEG (PSECT data) or PSECT bss pseudo-op is executed. For example, the first DSEG encountered sets the location counter to location zero in the Data Relative segment of memory. The following code is assembled in the Data Relative segment of memory. If a subsequent CSEG is encountered, the location counter will return to the next free location in the Code Relative segment and so on.

The ASEG, DSEG, CSEG pseudo-ops never have operands. If you wish to alter the current value of the location counter, use the ORG pseudo-op.

3.5 MACROS and Block Pseudo Operations

The macro facilities provided by Z80AS include three repeat pseudo operations: repeat (REPT), indefinite repeat (IRP), and indefinite repeat character (IRPC). A macro definition operation (MACRO) is also provided. Each of these four macro operations is terminated by the ENDM pseudo operation.

3.5.1 Terms

For the purposes of discussion of macros and block operations, the following terms will be used:

1. *dummy* is used to represent a dummy parameter. All dummy parameters are legal symbols that appear in the body of a macro expansion.

2. *dummylist* is a list of *dummy*s separated by commas.

3. *arglist* is a list of arguments separated by commas. *arglist* must be delimited by angle brackets. Two angle brackets with no intervening characters (<>) or two commas with no intervening characters enter a null argument in the list. Otherwise, an argument is a character or series of characters terminated by a comma or >. With angle brackets that are nested inside an *arglist*, one level of brackets is removed each time the bracketed argument is used in an *arglist*. A quoted string is an acceptable a rgument and is passed as such. Unless enclosed in brackets or a quoted string, leading and trailing spaces are deleted from arguments.

4. *paramlist* is used to represent a list of actual parameters separated by commas. No delimiters are required (the list is terminated by the end of line or a comment), but the rules for entering null parameters and nesting brackets are the same as described for *arglist*

3.5.2 REPT-ENDM

Syntax:

REPT *exp*
.
.
.
ENDM
The block of statements between REPT and ENDM is repeated *exp* times. *exp* is evaluated as a 16-bit unsigned number. If *exp* contains any external or undefined terms, an error is generated.

3.5.3 IRP-ENDM

Syntax:

IRP *dummy*,**<***arglist***>**
.
.
ENDM

The *arglist* must be enclosed in angle brackets. The number of arguments in the *arglist* determines the number of times the block of statements is repeated. Each repetition substitutes the next item in the *arglist* for every occurrence of *dummy* in the block. If the *arglist* is null (i.e., <>), the block is processed once with each occurrence of *dummy* removed. For example:

```
IRP     X,<1,2,3,4,5,6,7,8,9,10>
DB      X
ENDM
```

generates the same bytes as the REPT example.

3.5.4 IRPC-ENDM

Syntax:

IRPC *dummy*,[**<**]*string*[**>**]
.
ENDM

IRPC is similar to IRP but the *arglist* is replaced by a string of text and the angle brackets around the string are optional. The statements in the block are repeated once for each character in the string. Each repetition substitutes the next character in the string for every occurrence of *dummy* in the block. For example:

```
IRPC    X,0123456789
DB      X+1
ENDM
```

generates the same code as the two previous examples.

3.5.5 MACRO

Often it is convenient to be able to generate a given sequence of statements from various places in a program, even though different parameters may be required each time the sequence is used. This capability is provided by the MACRO statement.

The form is

MACRO *name dummylist*
.
.
ENDM

where *name* conforms to the rules for forming symbols and is the name that will be used to invoke the macro. The *dummy*s in *dummylist* are the parameters that will be changed (replaced) each time the MACRO is invoked. The statements before the ENDM comprise the body of the macro. During assembly, the macro is expanded every time it is invoked but, unlike REPT/IRP/IRPC, the macro is not expanded when it is encountered.

The form of a macro call is

name paramlist

where *name* is the name supplied in the MACRO definition, and the parameters in *paramlist* will replace the *dummy*s in the MACRO *dummylist* on a one-to-one basis. The number of items in *dummylist* and *paramlist* is limited only by the length of a line. The number of parameters used when the macro is called need not be the same as the number of *dummy*s in *dummylist*. If there are more parameters than *dummy*s, the extras are ignored. If there are fewer, the extra *dummy*s will be made null. The assembled code will contain the macro expansion code after each macro call.

> NOTE
>
> A dummy parameter in a MACRO/REPT/IRP/IRPC is always recognized exclusively as a dummy parameter. Register names such as A and B will be changed in the expansion if they were used as dummy parameters.

Here is an example of a MACRO definition that defines a macro called FOO:

```
MACRO FOO X
Y      DEFL   0
REPT   X
Y DEFL Y+1 DB
Y ENDM

ENDM
```

This macro generates the same code as the previous three examples when the call

```
FOO    10
```

is executed.

Another example, which generates the same code, illustrates the removal of one level of brackets when an argument is used as an arglist:

```
MACRO FOO  X
IRP    Y,<X>
DB     Y
ENDM
ENDM
```

When the call

```
FOO    <1,2,3,4,5,6,7,8,9,10>
```

is made, the macro expansion looks like this:

```
IRP    Y,<1,2,3,4,5,6,7,8,9,10>
DB     Y
ENDM
```

3.5.6 ENDM

Every REPT, IRP, IRPC and MACRO pseudo-op must be terminated with the ENDM pseudo-op. Otherwise a T error is generated at the end of each pass. An unmatched ENDM causes an O error.

3.5.7 EXITM

The EXITM pseudo-op is used to terminate a REPT/IRP/IRPC or MACRO call. When an EXITM is executed, the expansion is exited immediately and any remaining expansion or repetition is not generated. If the block containing the EXITM is nested within another block, the outer level continues to be expanded.

3.5.8 LOCAL

Syntax:

LOCAL *dummylist*

The LOCAL pseudo-op is allowed only inside a MACRO definition. When LOCAL is executed, the assembler creates a unique symbol for each *dummy* in *dummylist* and substitutes that symbol for

each occurrence of the *dummy* in the expansion. These unique symbols are usually used to define a label within a macro, thus eliminating multiply-defined labels on successive expansions of the macro. The symbols created by the assembler range from ??0001 to ??FFFF. Users will therefore want to avoid the term ??*nnnn* for their own symbols. If LOCAL statements are used, they must be the first statements in the macro definition.

3.5.9 Special Macro Operators and Forms

& The ampersand is used in a macro expansion to concatenate text or symbols. A dummy parameter that is in a quoted string will not be substituted in the expansion unless it is immediately preceded by &. To form a symbol from text and a dummy, put & between them.

For example:

```
MACRO   ERRGEN X
ERROR&X:PUSH    BC
LD      C,'&X'
JP      ERROR
ENDM
```

In this example, the call ERRGEN A will generate:

```
ERRORA: PUSH    BC
LD      C,'A'
JP      ERROR
```

;; In a block operation, a comment preceded by two semicolons is not saved as part of the expansion (i.e., it will not appear on the listing even under .LALL). A comment preceded by one semicolon, however, will be preserved and appear in the expansion.

! When an exclamation point is used in an argument, the next character is entered literally (i.e., !; and <;> are equivalent).

NUL NUL is an operator that returns true if its argument (a parameter) is null. The remainder of a line after NUL is considered to be the argument to NUL. The conditional IF NUL argument is false if, during the expansion, the first character of the argument is anything other than a semicolon or carriage return. It is recommended that testing for null parameters be done using the IFB and IFNB conditionals.

% The percent sign is used only in a macro argument. % converts the expression that follows it (usually a symbol) to a number in the current radix. During macro expansion, the number derived from converting the expression is substituted for the dummy. Using the % special operator allows a macro call by value. (Usually, a macro call is a call by reference with the text of the macro argument substituting exactly for the dummy.)

The expression following the % must conform to the same rules as the DS (Define Space) pseudo-op. A valid expression returning a non-relocatable constant is required.

In the example below, LB (the argument to MAKLAB) would normally be substituted for Y (the argument to MACRO) as a string. However, the % causes LB to be converted to a non-relocatable constant which is then substituted for Y. Without the % special operator, the result of assembly would be 'Error LB' rather than 'Error 1', etc.

```
MACRO MAKLAB  Y
ERR&Y:  DB     'Error &Y',0
ENDM
MACRO MAKERR  X
LB     DEFL    0
REPT    X
LB     DEFL    LB+1
MAKLAB  %LB
ENDM
ENDM
```

When invoked as MAKERR 3, the assembler will generate:

```
ERR1:   DB      'Error 1',0
ERR2:   DB      'Error 2',0
ERR3:   DB      'Error 3',0
```

3.6 Z80AS Errors

Z80AS errors are indicated by a one-character flag in column one of the listing file.
If a listing file is not being printed on the terminal, each erroneous line is also printed or displayed on the terminal. Below is a list of the Z80AS Error Codes:

A Too many IF statements. Maximum conditional nesting level reached.

B ENDIF without IF statement

C ELSE without IF statement

D Relative jump range error

E Expression error. Invalid operator, two consecutive operators, etc.

F Floating point value too big/small (overflow)

L Invalid identifier; Identifier contains invalid characters.

M Multiply Defined symbol

N Illegal opcode

O Bad opcode or objectionable syntax.
ENDM, LOCAL outside a block; DELF, EQU or MACRO without a name; bad syntax in an opcode; or bad syntax in an expression (mismatched parenthesis, quotes, consecutive operators, etc.).

P Phase error. Value of a Label or EQU name is different on pass 2.

Q Missing closing quote. An improperly closed string in a DB statement, etc.

R Relocation error.
Illegal use of relocation in expression, such as abs-rel. Data, code and COMMON areas are relocatable.

T Missing ENDM or ENDIF
Normally appears at the end of the listing, indicating an unterminated conditional or macro.

U Undefined symbol. A symbol referenced in an expression is not defined.

V Value error

W Symbol table overflow

Z Divide by zero
The expression being evaluated contains a division by zero.

3.7 Compatibility with ZAS assembler

Care has been taken to make Z80AS as compatible as possible with HiTech's ZAS assembler. There are, however, some differences, which are listed below:

- Z80AS limits the use of PSECT flags: only 'abs' flag is allowed, for the 'text' segment.
- Z80AS symbols can contain also the character @ , not allowed in ZAS
- Undefined symbols are treated as GLOBAL symbols and not reported as undefined.
- Cross reference output is not produced (ZAS uses the -C option for this)
- In Z80AS; arithmetic overflow in expressions is ignored. In ZAS, you must use the -N option for this, while in Z80AS this is the default mode (and the -N option is ignored).
- In Z80AS, -D command line option may be used to initialize with zero the DEFS buffers

3.8 Differences between ZSM4 assembler and Z80AS assembler

There are some differences, which are listed below:

- Command line syntax is different
- ZSM4 assembles Z80,Z180 and Z280 source files; Z80AS assembles only Z80 source files.
- ZSM4 builds .REL object files, compatible with Microsoft's L80 or Digital Research's LINK linkers; Z80AS builds .OBJ object files, compatible with HiTech's LINK linker.
- ZSM4 assembler allows using symbols up to 15 characters, only in uppercase, but stores only the first 6 characters into the object code; Z80AS allows using symbols in mixed upper / lowercase, up to 31 characters (compatible with HiTech's standards)
- MACRO definition is different (ZSM4: 'name MACRO params', Z80AS: 'MACRO name params')
- ORG: in Z80AS, ORG switches to the ASEG (PSECT text,abs) and sets the PC=value, while in ZSM4, ORG stays in the current segment and sets the PC=value
- Z80AS does not support the following ZSM4 pseudo operators: ASET, COMMON, COMMENT, DEFL, DEPHASE, ENTRY, EXTRN, IDENT, IFZ180, IFZ280, MACLIB, NAME, PHASE, PUBLIC, REQUEST, RQST, .Z80, .Z180, .Z280, .EVEN, .ODD