ZSM4

Macro Assembler

for CP/M-80, RSX180 and UZI180 Operating Systems

Reference Manual

Version 4.1

Hector Peraza, April 2019

Table of Contents

1 Introduction

ZSM4 is a macro-assembler for microcomputer systems with a Z80 or compatible CPU with a CP/M or RSX180 operating system. 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.

ZSM4 main features are:

- Assembles Z80, Z180 or Z280 code using the standard Zilog/Mostek mnemonics.
- Supports conditional assembly, enhanced by an expanded set of conditional pseudo operations that include testing of assembly pass, symbol definition, and parameters to macros. 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 REL file suitable for linking with Microsoft's L80 or Digital Research's LINK.

2 Running ZSM4

2.1 Invoking the assembler

The command to run ZSM4 is

`ZSM4` [*`command`*]

The format of the command is explained in the next section. If no command is specified, the assembler enters interactive mode, indicated by the prompt "*", indicating it is ready to accept commands.

2.2 Command format

A command to ZSM4 has the following format:

[*`objfile`*],[*`lstfile`*]=*`srcfile`*[*`/option`*][*`/option...`*]

where *objfile*, *lstfile* and *srcfile* are file specifications for the output object file, output listing file and input source file respectively. All file names should follow the operating system's conventions for file names and extensions, and can specify also a device name.

The default extensions are as follows:

File	CP/M	RSX180
Relocatable object file	REL	OBJ
Listing file	PRN	LST
Source file	MAC	MAC

The default device name with the CP/M operating system is the currently logged disk. The default device name with the RSX180 operating system is SY0:, the default user device.

The device names are as follows:

Device	CP/M	RSX180
Disk drives	`A:, B:, C:`	`SY:, LB:, DY`*`n`*`:, DU`*`n`*, etc.
Line printer	`LST:` or `PRN:`	`LP:`
Console	`TTY:` or `CON:`	`TI:`

Examples:

`,TTY:=TEST`	Assemble the source file TEST.MAC and list the program on the console. No object code is generated. Useful for error check.
`SMALL,LST:=B:TEST`	Assemble TEST.MAC (found on disk drive B), place the object file in SMALL.REL and list the program on the line printer.

2.2.1 Option switches

Options are specified via a *switch* consisting of a slash followed by a single letter. Certain options require an additional argument, as described below. More than one switch can be used, but each must be preceded by a slash. All switches are optional.

The available switches are:

/L Force generation of a listing file.

/D*symbol*[=*value*] Define symbol and optionally assign a value. The value is a numeric constant following the format described in Section 2.3.3. If no value is specified, 0 is assumed.

/S*n* Set the maximum symbol name length in the REL file. The value of *n* can be anywhere from 5 to 8, and defaults to 6.

/Z*n* Select the initial target CPU type, *n* can be 0 for Z80 (default), 1 for Z180 and 2 for Z280. This option can be overridden by a .Z80, .Z180 or .Z280 pseudo-operator (see Section 2.5.25)

/M Initialize block data areas defined by the DEFS or DS pseudo-op to zeros. If the switch is not specified, the space is not guaranteed to contain zeros.

/U Treat all undefined symbols as External.

Examples:

`=TEST/L`	Assemble TEST.MAC, place the object file in TEST.REL and a listing file in TEST.PRN
`LAST=TEST/U`	Assemble TEST.MAC and place the object file in LAST.REL. No listing file will be generated. All undefined symbols will be declared as External.

2.3 Source file format

Input source lines of up to 160 characters in length are acceptable. ZSM4 preserves lower case letters in quoted strings and comments. All symbols, opcodes and pseudo-opcodes typed in lower case will be converted internally to upper case.

2.3.1 Statements

Source files input to ZSM4 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 can be immediately followed by a colon. If the label is the only item on a line, then the colon is mandatory. If the label is followed by two colons, it is declared as PUBLIC (see Section 2.5.11). For example:

```
FOO::   RET
```

is equivalent to

```
PUBLIC  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. Extended comments can be entered using the .COMMENT pseudo operation (see Section 2.5.22).

2.3.2 Symbols

ZSM4 symbols may be of any length, however, only the first 12 characters are significant, and in addition only the first six are stored in the object file (unless changed with the S option switch, see Section 2.2.1). The following characters are legal in a symbol:

```
A-Z 0-9 $ . ? _ @
```

A symbol may not start with a digit. When a symbol is read, lower case is translated into upper case. If a symbol reference is followed by ## it is declared external (see also the EXTRN pseudo-op, Section 2.5.13).

2.3.3 Numeric Constants

The default base for numeric constants is decimal. This may be changed by the .RADIX pseudo-op (see Section 2.5.24). 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.

2.3.4 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.

2.4 Expression Evaluation

2.4.1 Arithmetic and Logical Operators

The following table list the allowed operators and their precedence:

Operator	Function	Precedence
NUL	Test for Null argument	1
LOW	Take LOW byte	2
HIGH	Take HIGH byte	2
*	Unsigned Multiply	3
/	Unsigned Divide	3
MOD	Unsigned Module	3
SHR	Shift Right	3
SHL	Shift Left	3
-	Unary Minus	4
+	Add	5
-	Subtract	5
EQ	Equal	6
NE	Not Equal	6
LT	Less Than	6
LE	Less Than or Equal	6
GT	Greater Than	6
GE	Greater Than or Equal	6
LESS	Signed Less Than	6
NOT	Bitwise Not	7
AND	Bitwise And	8
OR	Bitwise Or	9
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.

2.4.2 Modes

All symbols used as operands in expressions are in one of the following modes: Absolute, Data Relative, Program (Code) Relative or COMMON. (See Section 2.5 for the ASEG, CSEG, DSEG and COMMON pseudo-ops.) Symbols assembled under the ASEG, CSEG (default), or DSEG pseudo-ops are in Absolute, Code Relative or Data Relative mode respectively.

The number of COMMON modes in a program is determined by the number of COMMON blocks that have been named with the COMMON pseudo-op. Two COMMON symbols are not in the same mode unless they are in the same COMMON block. 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

```
F00 + (BAZ - ZAZ)
```

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

2.4.3 Externals

Aside from its classification by mode, a symbol is either External or not External. (See EXTRN, Section 2.5.13.) 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.

2.5 Pseudo operators

2.5.1 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. The effect of ASEG is also achieved by using the code segment (CSEG) pseudo operation and the [P] switch in DR's LINK. See also Section 2.5.30.

2.5.2 COMMON

Syntax:

COMMON */blockname/*

COMMON sets the location counter to the selected common block in memory. The location is always the beginning of the area so that compatibility with the FORTRAN COMMON statement is maintained. If *blockname* is omitted or consists of spaces, it is considered to be blank common. See also Section 2.5.30.

2.5.3 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 (see also Section 2.5.30.)

2.5.4 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 counter.

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.

2.5.5 DEFC, DC (Define Character)

Syntax:

```
DC    string
DEFC  string
```

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.

2.5.6 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. If a U error is not generated during pass 2, a phase error will probably be generated because the DS generated no code on pass 1.

2.5.7 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.

2.5.8 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.

2.5.9 DSEG

Syntax:

```
DSEG
```

DSEG sets the location counter to the Data Relative segment of memory. The location of the data relative counter wil be that of the last DSEG (default is 0), unless an ORG is done after the DSEG to change the location. See also Section 2.5.30.

2.5.10 END

Syntax:

```
END   [exp]
```

The END statement specifies the end of the program. If *exp* is present, it is the start address of the program. If *exp* is not present, then no start address is passed to the linker for that program.

2.5.11 ENTRY, GLOBAL, PUBLIC

Syntax:

```
ENTRY   name[,name...]
GLOBAL  name[,name...]
PUBLIC  name[,name...]
```

ENTRY, GLOBAL or PUBLIC declares each name in the list as internal and therefore available for use by this

program and other programs to be loaded concurrently. All of the names in the list must be defined in the current program or a U error results. An M error is generated if the name is an External name or COMMON block name.

2.5.12 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.

2.5.13 EXT, EXTRN

Syntax:

```
EXT     name[,name...]
EXTRN   name[,name...]
```

EXT or EXTRN declares that the name(s) in the list are external (i.e., defined in a different program). If any item in the list references a name that is defined in the current program, an M error results. A reference to a name where the name is followed immediately by two pound signs (e.g., NAME##) also declares the name as external.

2.5.14 INCLUDE, MACLIB

Syntax:

```
INCLUDE filename
MACLIB  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. INCLUDE and MACLIB are synonymous.

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 ZSM4 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.

2.5.15 NAME

Syntax:

```
NAME    ('modname')
NAME    'modname'
```

NAME defines a name for the module. Only the first six characters are significant in a module name. A module name may also be defined with the TITLE pseudo-op. In the absence of both the NAME and TITLE pseudo-ops, the module name is created from the source file name.

2.5.16 IDENT

Syntax:

```
IDENT   'ident'
```

IDENT defines an identification string for the module that can be used for version tracking. Only the first six characters are significant.

2.5.17 ORG (Define Origin)

Syntax:

```
ORG     exp
```

The location counter is set to the value of *exp* and the assembler assigns code to memory locations starting with that value. All names used in *exp* must be known on pass 1, and the value must either be absolute or in the same area as the location counter.

2.5.18 EJECT, FORM, PAGE

Syntax:

```
PAGE    [exp]
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.

2.5.19 DEFL, ASET

Syntax:

```
name ASET exp
name DEFL exp
```

DEFL is the same as EQU, except that no error is generated if *name* is already defined. DEFL and ASET are synonymous.

2.5.20 SUBTTL

Syntax:

```
SUBTTL  text
```

SUBTTL specifies a subtitle to be listed on the line after the title (see TITLE, Section 2.5.21) on each page heading. *text* is truncated after 60 characters. Any number of SUBTTLs may be given in a program.

2.5.21 TITLE

Syntax:

```
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. The first six characters of the title are used as the module name unless a NAME pseudo operation is used. If neither a NAME or TITLE pseudo-op is used, the module name is created from the source file name.

2.5.22 .COMMENT

Syntax:

```
.COMMENT delimiter...text...delimiter
```

The first non-blank character encountered after .COMMENT is the delimiter. The following *text* comprises a comment block which continues until the next occurrence of *delimiter* is encountered. For example, using an asterisk as the delimiter, the format of the comment block would be:

```
.COMMENT  *
any amount of text entered
here as the comment block
.
.
.      *
;return to normal mode
```

2.5.23 .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 ZSM4 is in pass 2.

2.5.24 .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.

2.5.25 .Z80, .Z180, .Z280

Syntax:

```
.Z80
.Z180
.Z280
```

.Z80 enables the assembler to accept Z80 opcodes. This is the default condition. Z80 mode may also be set by appending the /Z0 switch to the ZSM4 command string (see Section 2.2.1.)

.Z180 enables the assembler to accept Z80/HD64180 opcodes. Z180 mode may also be set by appending the /Z1 switch to the ZSM4 command string.

.Z280 enables the assembler to accept Z280 opcodes. Z280 mode may also be set by appending the /Z2 switch to the ZSM4 command string.

2.5.26 RQST, .REQUEST

Syntax:

```
RQST     filename[,filename...]
.REQUEST filename[,filename...]
```

.REQUEST sends a request to the linker loader to search the file names in the list for undefined globals. The file names in the list should be in the form of legal symbols. They should not include file name extensions or disk specifications. Normally, the linker supplies a default extension and assumes the default disk drive.

2.5.27 .EVEN and .ODD

Syntax:

```
.EVEN
.ODD
```

.EVEN causes the assembler to emit a null byte (NOP instruction) if the current address is odd, so the next instruction or data byte will be aligned to a word boundary (even address).

Likewise, .ODD causes the assembler to emit a null byte (NOP instruction) if the current address is even, so the next instruction or data byte will be aligned to an odd address.

Note that for .EVEN and .ODD to work correctly at run time, the corresponding module must be loaded by the linker on an even address.

2.5.28 Conditional Pseudo Operations

The conditional pseudo operations are:

IF/IFT *exp*	True if *exp* is not 0.
IFF *exp*	True if *exp* is 0.
IF1	True if pass 1.
IF2	True if pass 2.
IFZ80	True if the assembler is in Z80 mode (see Section 2.5.25.)
IFZ180	True if the assembler is in Z180 mode (see Section 2.5.25.)
IFZ280	True if the assembler is in Z280 mode (see Section 2.5.25.)
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:

```
IFxx    [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.

2.5.28.1 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.

2.5.28.2 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.

2.5.29 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	Suppresses the listing until a LIST ON command.
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. This is the default condition.
NOSORT	Do not sort the symbol table. This typically results in shorter assembly times.
MACROS	List the complete macro text for all MACRO/REPT/IRP/IRPC expansions.
NOMACROS	Supress 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.

For compatibility with RMAC and M80, 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

2.5.30 Relocation Pseudo Operations

ZSM4 supports all four relocatable areas defined by the REL object code relocatable format: ASEG (Absolute), CSEG (Code), DSEG (Data) and named COMMONs.

The default mode for the assembler is 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 or DSEG or COMMON 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.

2.5.30.1 ORG Pseudo-op

At any time, the value of the location counter may be changed by use of the ORG pseudo-op. The form of the ORG statement is:

ORG *exp*

where the value of *exp* will be the new value of the location counter in the current mode. All names used in *exp* must be known on pass 1 and the value of *exp* must be either Absolute or in the current mode of the location counter. For example, the statements

```
DSEG
ORG     50
```

set the Data Relative counter to 50, relative to the start of the Data Relative segment of memory.

2.5.31 Relocation Before Loading

Two pseudo-ops, .PHASE and .DEPHASE, allow code to be located in one area, but executed only at a different, specified area.

For example:

```
0000'                           .PHASE  100H
0100    CD 0106       FOO:      CALL    BAZ
0103    C3 0007'                JP      ZOO
0106    C9            BAZ:      RET
                                .DEPHASE
0007'   C3 0005       ZOO:      JP      5
```

All labels within a .PHASE block are defined as the absolute value from the origin of the phase area. The code, however, is loaded in the current area (i.e., from 0000' in this example). The code within the block can later be moved to 100H and executed.

2.6 MACROS and Block Pseudo Operations

The macro facilities provided by ZSM4 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.

2.6.1 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* (see example, Section 2.6.5.) A quoted string is an acceptable argument 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* (see example, Section 2.6.5.)

2.6.2 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. Example:

```
X     DEFL  0
      REPT  10    ;generates DB 1 - DB 10
X     DEFL  X+1
      DB    X
      ENDM
```

2.6.3 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.

2.6.4 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.

2.6.5 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

```
name   MACRO   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:

```
FOO     MACRO   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:

```
FOO     MACRO   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
```

2.6.6 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.

2.6.7 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.

2.6.8 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.

2.6.9 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:

```
ERRGEN  MACRO   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.

```
MAKLAB  MACRO   Y
ERR&Y:  DB      'Error &Y',0
        ENDM
MAKERR  MACRO   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
```

2.7 ZSM4 Errors

ZSM4 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 ZSM4 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.

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.

2.8 Compatibility with other assemblers

Care has been taken to make ZSM4 as compatible as possible with Microsoft's M80 and Digital Research's RMAC. There are, however, some differences, which are listed below:

- The .TFCOND pseudo-operator (toggle listing of false conditionals) is not implemented.
- ZSM4 allows several TITLE pseudo-operators to appear in the source file, while M80 allows only one.
- If no NAME pseudo-operator is specified, M80 uses the first word of the TITLE statement as the default module name, whereas ZSM4 uses the source file name.
- ZSM4 stores up to 8 characters of symbol names in REL files (see the /S*n* option switch in Section 2.2.1), whereas both M80 and RMAC store up to 6.
- ZSM4 does not support REL format extensions like M80 3.44 or SLR do. Only few linkers support them, anyway.

- The byte order of character constants comprised of two characters is the same as in RMAC, but in M80 is the opposite (see Section 2.3.3). Thus, a statement like

```
LD      HL,'AB'
```

  will load the L register with 'A' and H with 'B'.

2.9 Additional notes

The Z280 mode has a few caveats:

- The operand of instructions that accept either an 8- or 16-bit value or displacement must be know on pass 1. Examples of such instructions are:

```
ADD     A,(IX+d8)       and     ADD     A,(IX+d16)
LD      (HL),d8         and     LD      (HL),d16
etc.
```

  That's because the different forms of the same instruction generate different opcodes and/or have different length. If the operand is not known on pass 1, then the assembler will have to guess or to chose arbitrarily one of the forms; when the operand size is finally know on pass 2, then a phase error will occur if its size does not match the one guessed during pass 1. In order to eliminate such restriction, the assembler would have to perform additional passes over the source code, becoming more complicated and slower.

- As described in the Z280 Manual, the syntax of the four Extended Instructions EPUF, EPUI, EPUM and MEPU is not clear. The instructions require a four-byte template argument which is loaded into, or decoded by, the co-processor and the Z280 Manual even specifies a format for it but nowhere is a description or an example of how the instruction mnemonic actually looks like. Even when the template is destined for the co-processor, the Z280 CPU may decode part of it. In particular, the number of bytes to be transferred by the EPUM and MEPU instructions is extracted from the fourth template byte, and thus it looks like the template ought to be part of the instruction mnemonic.

  In this version of ZSM4, the template bytes are specified separately from the instruction and therefore is up to the user to follow the appropriate format. For example:

```
EPUM    (HL)
DEFB    t1,t2,t3,t4
```

  where *t1*, *t2*, *t3* and *t4* are the template bytes.

- The .EVEN and .ODD alignment pseudo-operators (see section 2.5.27) require appropriate linker support. Specifically, any code or data segment using these pseudo-operators must start on an even address boundary. If the linker has no provision for such feature, then the same result can be achieved by making all segments to be linked of even size, and linking the final program starting on an even address. To ensure that all segments have an even size, end them with an .EVEN pseudo-operator.

A Operation under CP/M

The ZSM4 CP/M executable has a size of nearly 22 Kbytes, and therefore a system with large TPA is recommended for running the assembler, especially if large source files are to be assembled or if the sources use a large number of MACROs. On the plus side, the symbol table format is much more compact than that of e.g. M80.

File specifications follow the standard CP/M convention:

```
drive:filename.ext
```

Only *filename* is required, all other fields are optional and default to the following values:

drive	the current drive
ext	MAC for the source, REL for the object output and PRN for the listing file

If no command line is present, the assembler will enter interactive mode and prompt for a command line:

```
>ZSM4
Z80/Z180/Z280 Macro-Assembler V4.1

*
```

In interactive mode several commands can be entered to process several files without having to reload the assembler every time.

Under CP/M 3.0 and MP/M, the date and time of assembly is shown on the listing output.

B Operation under RSX180

ZSM4 is built with a default task name of `...MAC`; a different name can be specified in the INStall command, for example:

```
INS ZSM4/TASK=...ZSM
```

The task is also built with a default memory increment amount of 8000 bytes, which should be enough for assembling small to medium-sized files. However, for large files specifying a larger memory increment via the INStall command is *not* necessary, since the assembler can request automatically more memory from the system if/as it becomes necessary. If the request is not granted, and error will be displayed.

File specifications follow the standard RSX180 convention:

dev:[*directory*]*filename*.*ext*;*vers*

Note that the square brackets above are part of the syntax and therefore required when specifying a directory. Only *filename* is required, all other fields are optional and default to the following values:

dev	SY0: (the user's login device)
directory	the current user's directory
ext	`MAC` for the source, `OBJ` for the object output and `LST` for the listing file
vers	highest (latest) version of the file.

If the task has been installed, it can be invoked by entering its installed task name, e.g.

MAC [*command*]

when not installed, the task can be invoked using the RUN command, for example:

```
RUN $ZSM4
```

If no command line is present, the assembler will enter interactive mode and prompt for a command line:

```
>MAC
MAC>
```

ZSM4 also supports indirect command files under RSX180 the standard way:

MAC @*filespec*

where *filespec* is a valid command file specification. If no extension is specified, CMD is assumed. The command file simply contains a list of commands to be executed by ZSM4, one per line, and can in turn include calls to additional command files up to a nesting level of 3.

C Operation under UZI180

Since UZI180 has an embedded CP/M emulator, the CP/M version of ZSM4 can be run directly from the UZI180 shell prompt. The usual limitation of the 8.3 CP/M filename format applies.

A native UZI180 version of ZSM4 is currently under development.

D Operation under Linux or Windows

The CP/M version of ZSM4 can be used for cross-assembly on a Linux or Windows system by using John Elliott's zxcc program (http://www.seasip.demon.co.uk/Unix/Zxcc/index.html). Zxcc emulates both a Z80 and a CP/M environment in a way such that a CP/M application can be run from the Linux or Windows command prompt as if it was a native application.

Since ZSM accepts source files with both CP/M and Unix end-line conventions, no conversion to the CP/M format is necessary.

Being a CP/M program, a few restrictions apply:

- File names are limited to the 8.3 format, and in lowercase.
- If the source files INCLUDE additional files (see Section 2.5.14), then the zxcc command invoking ZSM4 must be run in the directory where the source files are.
- Options switches must be escaped properly (see the zxcc documentation for details), or else the Unix shell may interpret them as directory specifications.

Examples:

```
zxcc zsm4 test,test=test

zxcc zsm4 -"=test/L"
```

Note how the whole command in the second example is escaped, since it contains the /L option switch.

Zxcc allows invoking ZSM4 from a Makefile, for example:

```
.PREFIX:
.PREFIX: .mac .rel

SRCS = main.mac functions.mac misc.mac
OBJS = $(SRCS:.mac=.rel)

all: example.com

$(OBJS): %.rel: %.mac *.inc
        zxcc zsm4 -"=$</l"

example.com: $(OBJS) mylib.lib
        zxcc link -"$@=main,functions,misc,mylib.lib[s]"
```