Reference Guide

[M]ultiple [P]rocessor [M]odule Assembler - Z80 Edition V1.4, 2013
Reference Guide

This draft document is currently being finalized for V1.4 - migrated from V1.2 (unfinished) html

Instructing the assembler on the command line

Mpm is executed using the operating system standard command line environment (CLI).
When executing the assembler from the command line without options, i.e. mpm<ENTER>, a help page is automatically displayed where after control is returned to the operating system command line. The page shortly explains the syntax of option specification and which options are available.

The syntax of the assembler options is a straightforward design. Filenames or a project file are always specified. The options may be left out (surrounded here with []):
`mpm [-options] {filenames} | @projectfile`

As seen above [-options] must be specified first, followed by the names of assembler source files. You either choose to specify all file names or a @<projectfile> containing all file names. Filenames may be specified with or without the .asm filename extension. At least on source file must be specified and may be repeated with several file names. Only one project file may be specified if no source file names are given. You may use comment lines inside projects files using semicolon.

Options are used to control the assembly process and output. They are recognized by the assembler when you specify a leading minus before the option identifier ('-'). When the assembler is executed options are all preset with default values and are either switched ON or OFF (active or not).

Upper and lower case letters are distinguished, which means that 'a' is a different command than 'A'. If an option is to be turned off, a 'n' is specified before the identification, e.g. **`-nl`**. Single letter options without parameters can be concatanated in a single option block.

Default options are: **`-sm -nvadbctgA`**

Options without parameters (options are specified with **-** in front of option, for example **-v**)	
v	Verbose output during assembly. Not enabled by default.
t	Create listing file output. Listing file contains generated machine code output combined with source code lines and optionally a symbol table. Not enabled by default.
s	Create symbol table in listing file, or if no listing file enabled - then create a standalone symbol table file. Enabled by default.

m	Create address map information file (only when performing linking of object files). Enabled by default.
d	Assemble only updated source files. Not enabled by default (always compile even if object file exists).
b	Link object files into executable binary. Not enabled by default.
a	Assemble only updated source files and automatically link object files into executable binary. Not enabled by default.
c	Split linked executable into 16K binary files (using .bnX extension). Not enabled by default.
g	Create global address definition file (dump all referenced XREF names). Not enabled by default.
A	Address align DEFW & DEFL constants. Not enabled by default.
C	Override compile error/warnings line numbers using external source code line number reference (specify external line numbers with LINE directive in souce code). Implemented as "C line mode" when Mpm functions as back end for z88dk C compiler. Not enabled by default.
R	Generate address independent executable binary (code needs to run in RAM). Not enabled by default.
plus	INVOKE directive generates RST 28H instruction opcode (otherwise CALL instruction). Not enabled by default.
IXIY	All IY related instructions are generated with IX instruction opcodes and vice versa. Not enabled by default.
crc32	Creates a *.crc file that contains the CRC32 (32bit) hexadecimal value of the compiled binary. Not enabled by default.

Options with parameters (options are specified with **-** in front of option, for example **-r**<address>)	
D<symbol name>	Define symbol name as logically TRUE and available for current compilation session.
r<address>	Re-define the ORG linking executable start address (override ORG directive in first source code file).
e<ext>	Replace default "asm" input source file name extension with <ext>. Max 3 letters (specified without ".")
M<ext>	Replace default "obj" filename extension for output object module files with <ext>. Max 3 letters (specified without ".")
o<filename>	Override generated executable binary default filename with <filename>. The default filename is derived from the first specified source filename or derived from the project filename. Also requires -b or -a option (to create a linked executable binary).
I<path>	add INCLUDE directive search path (relative paths names are allowed). Several paths may be concanated by using operating system path separator (';' in Windows, ':' in Unix). Avoid space between the path and the separator. -I option may be specified multiply times at command line.
L<path>	add LIB library file search path (relative paths are allowed). Several paths may be concanated by using operating system path separator (';' in Windows, ':' in Unix). Avoid space between the path and the separator. -L option may be specified multiply times at command line.
x<filename>	Create library file from specified modules. Use with a @projectfile specification for convenience. Using -b or -a with this option is meaningless.
l<filename>	Statically link found library modules in specified library file, which is appended to compiled code that used the LIB directive. Paths in filename is optional. May be combined with the -L option (to search for library at search paths).

Environment variables

The assembler looks into the operating system environment for these three variables. Convenient for shell scripts where the environment variables are instantiated for use while the shell is open:

MPM_Z80_INCLUDEPATH	Defines the default path of header/include definition files. Multiple paths may specifed using operating system specific path separator (see options above). Equivalent to the -I option
MPM_Z80_LIBPATH	Defines the default seach path for Mpm library files. Multiple paths may specifed. Multiple paths may specifed using operating system specific path separator (see options above). Equivalent to the -L option.
MPM_Z80_STDLIBRARY	Defines the default filename of the standard library to be scanned by LIB directives.

MPM module assembler file types

The MPM Module Assembler uses several different file name extensions to distinguish the type of files processed and generated. All extensions are using three letter combinations. All generated files will replace the 'asm' extension with the appropriate default output extension (keeping remaining input filename intact).

asm	The source file (input) extension. May be overridden using the -e option.
obj	The generated object module extension. The file contains intermediate generated machine code, optional address origin, symbols and expressions. May be overridden using the -M option.
err	Error reporting files. If any errors should occur, they will be written to this file containing information of where the error occurred in the corresponding source file.
lst	Listing file. This file contains a hexadecimal output column of the generated machine code that corresponds to the Z80 mnemonic instruction or directive, followed by a copy of the original source code, line by line. A symbol table is dumped at the end of the listing file.
sym	Symbol Table file. This file contains a symbol name and hexadecimal value output columns of the found and used symbol definitions in the sources. The symbol table file is automatically generated if no listing file was enabled. This file is generated by default.
bin	The final output file containing the executable linked and address relocated object modules. The complete filename (including extension) may be overridden using the -o option.
map	This file contains a list of all defined address labels from all linked/relocated modules (and optionally LIB modules) with their calculated (absolute) address when the code is loaded into the executing memory environment. All labels are sorted in two sections; first by name, then by address, both in ascending order.
crc	This file contains a calculated CRC32 hexadecimal value of the final executable output file. It is only generated if you use the -crc32 co mmand line option.
def	This file contains a DEFC directive list of all globally declared address labels with their calculated (absolute) origin address, fetched only during assembly of source file modules. The file may be useful for cross referencing in separate compile projects.
lib	Library file, which may be created using the -x option. Library modules may be included into application code during linking of object modules using the -l option.

The standard platform identifier

When the MPM Assembler is running, a standard platform identifier (with value 1) is made available when typically using IF directives or other expressions. The identifier reflects the name of the platform:

"MSDOS" - the assembler is running on the MS-DOS or Windows platform of computers
"UNIX" - the assembler is running on the LINUX/UNIX/MAC OSX platform.

The identifier might be very handy when you need to port your source files to different computer platforms.

Source file structure and layout

The composition of a source file module is completely free to the programmer. How he chooses to place the source code on a text line has no effect of the parsing process of the assembler. The linefeed interpretation is also handled by MPM - it understands all known line ending formats:

<LF> (used by UNIX/AMIGA/MAC OS X)
<CR><LF> (used by MSDOS, WINDOWS)
<CR> (used by Z88, Mac OS 8/9)

Each source file begins with a MODULE definition, then optionally an ORG directive followed by INCLUDE directives (importing constant definitions). Also appropriate is to specify all global and external identifiers (for cross-module referencing) at the top of the source file, using the XDEF and XREF directives. Good practice is also to specify in which module (file) the XREF identifier is created (where it is XDEF'ined). The rest of the file contains the actual code or data.

Comments in source files

Comments in Z80 source files are added using a semicolon, followed by the text of the comment. When the assembler meets a semicolon the rest of the current source line is ignored until the linefeed. The semicolon may be placed anywhere in a source line. The following is an example on how to use comments in Z80 source files:

```
; **********************
; main menu
;
mainmenu: xor a
          ret    ; back to calling routine...
```

You can also use multi-line comments like used in C source code:

```
/*
   main menu

   this is the main application menu
*/
mainmenu: xor a
          ret    ; back to calling routine...
```

Defining symbolic address labels

The main reason for using an assembler is to be able to determine symbolical addresses and use them as reference in the code, instead of absolute addresses. These are defined by a name preceded with a full stop or ending with a colon. It is allowed to place an instruction mnemonic or directive after an address label on the same line. An address label may be left as a single statement on a line - you may of course use comment after the label:

```
mainmenu:   xor a
.mainmenu2  ; another comment
```

It is not allowed to position two label definitions on the same line. However, you may place as many label definitions after each other on separate lines - even though no code is between them. They just receive the same assembler address during compilation. It is not possible to use two identical address labels in the same source file. All labels and constant names are case insensitive.

Using expressions

Expressions is a necessary tool in source files. They define and explain things much clearer than just magic numbers. The MPM Module Assembler allows expressions wherever a parameter is needed, even when a constant is defined. This applies to Z80 mnemonic instructions, directives and even in character strings. The resulting value from an evaluated expression is always an integer. All expressions are calculated as 32 bit signed integers. However, the parameter type defines the true range of the expression. For example, you cannot store a 32 bit signed integer in an 8 bit LD instruction like LD A, n.

If a parameter is outside an allowed integer range an assemble error is reported. Expressions may be formed as arithmetic and relational expressions. Several components are supported: symbol names (identifiers), constants, ASCII characters and all the usual suspects of arithmetic operators.

Numeric constants

Apart from specifying decimal integer numbers, you are allowed to use hexadecimal constants, binary constants and ASCII characters. The following definitions are available:

Digits without leading or trailing specifiers are interpreted as decimal constants.

Decimal	456567, 456d
Hexadecimal	$8000, 0xC000 (49152), or 4000h
Binary	@11000000 (192d) or 1011b (11d)
Ascii	'A' (65d)

Arithmetic operators

All basic arithmetic operators are supported. The following table shows them all. Result of example is shown in brackets for clarity of operator.

Operator symbol	Function	Example
+	Addition	12+13
-	Unary minus, subtraction	-10, 12 - 45
*	Multiplication	45 * 2 (90)
/	Division	256 / 8 (32)
>	Unary division by 256	>1024 (4)
%	Modulus	256%8 (0)
<	Unary modulus by 256	<65000 (232)
**	Power	2 ** 7 (128)
&	Binary AND	255 & 7 (7)
|	Binary OR	128 | 64 (192)
:	Binary NOR	128 : 128 (0)
^	Binary XOR	12 ^ 3 (15)
~	Unary Binary NOT	~128 (127)
<<	Binary left shift	2 << 4 (32)
>>	Binary right shift	128 >> 4 (8)

Arithmetic operators use the standard operator precedence, shown from highest to lowest:

0x $ @ ' (constant identifiers)
() []
~ > <
**
* / %
+- | ^ & : << >>

If you want to override the default operator precedence rules, use brackets () or []. The square bracket have been implemented as an alternative to avoid ambiguity with Z80 instruction mnemonics where indirect addressing uses the () as part of their syntax.

The # operator

This is not a real operator, though. It is only used as the first symbol in an expression to identify that this expression evaluates to a constant, i.e. containing no relocatable information. This feature may be necessary when you have to calculate a distance between two address labels without letting the assembler think it is a relocatable address. The assembler thinks that expressions containing address labels are relocatable items, which would create unwanted side effects during linking of object modules. However, since the assembler cannot see this as a constant expression, you have to force it as one.

```
LD BC, end_relocator-relocator  ; Mpm adds an ORG constant during
linking
LD BC, #end_relocator-relocator ; Mpm sees a constant expression
```

Relational operators

With relational operators you may form logical expressions resulting in true or false conditions. The resulting value of a true expression is 1. The resulting value of a false expression is 0. These operators are quite handy when you need to perform complex logic for conditional assembly in IF-ELSE-ENDIF conditional assembly. The following relational operators are available:

= equal to
<> not equal to
< less than
> larger than
<= less than or equal to
>= larger than or equal to
! not

You may link several relational expressions with the binary operators AND, OR, XOR and NOT.
It is perfectly legal to use relational expressions in parameters requiring an arithmetic value. For example:

```
LD A, [USING_IBM = 1] | RTMFLAGS    ; use [] to ensure LD A,n
instruction opcode
```

Predefined assembler variables

Apart from declaring your own constants, Mpm has a few pre-declared constants that might become useful in your programs.

Program counter

On occasional circumstances it may be necessary to use the current location of the assembler program counter in an expression e.g. calculating a relative distance. This may be done with the help of the $PC identifier. An example:

```
errmsg0: DEFM errmsg1 - $pc - 1, "File open error"
errmsg1: DEFM errmsg2 - $pc - 1, "Syntax error"
errmsg2:
```

Here, a length byte of the following string (excluding the length byte) is calculated by using the current $PC address value.

System time

When Mpm is starting a compile session, the system clock is fetched once and made available as integer variables:

$YEAR	4-digit year value, eg. 2006
$MONTH	Month of the year, range is 1 to 12.
$DAY	The day of the month, range is 1-31.
$HOUR	Range is 0-23.
$MINUTE	Range is 0-59.
$SECOND	Range is 0-59.

Directive reference

The MPM Module Assembler directives are used to manipulate the Z80 assembler mnemonics, generate data structures, variables and constants - even including binary files while code generation is performed.

As the name imply they direct the assembler to perform other tasks than just parsing and compiling Z80 instruction mnemonics. All directives are treated as mnemonics by the assembler, i.e. it is necessary that they appear as the first command identifier on the source line (NOT necessarily the first character). Only one directive is allowed per single source line. As with Z80 mnemonics, directives are case insensitive.

The following syntax is used to describe the directives:

<> defines a syntactic entity, i.e. a number, character or string.
{} defines a repetition of a syntactic entity.
[] defines an option that may be left out.

ALIGN

ALIGN <no of bytes>
Align (or zero-pad) a specified number of bytes relative to the current PC. Argument value range = 1-16. For example use ALIGN 4 to ensure that the new PC is located at a module address that is aligned in 4 byte entities. Another example would be to use ALIGN 2 to make sure that the next PC address is located at an even address. ALIGN only aligns addresses on the current module compilation - not on the entire scope of the linked modules. If you want to align all linked modules to even adresses, use an ALIGN at the end of each module.

ASCII, ASCIIZ, DM, DMZ, DEFM, DEFMZ

ASCII <string expression>
ASCIIZ <string expression>
DM <string expression>
DMZ <string expression>
DEFM <string expression>
DEFMZ <string expression>

All these directives is the same thing; to store a string at current PC.

Strings are enclosed in double quotes. Strings may be concatenated with byte constants using '.' or ','.
This is useful if control characters (or even arithmetic expressions resulting into an 8bit value) need to be a part of the string and cannot be typed from the keyboard.

All directive names ending with a Z indicates that the end of the string is automatically null-terminated (appended with a 0 byte).

Example:

```
ASCIIZ "This is a title", 13, 10
DEFM "(c) Zsoft 1995-2006", 13, 10, 'A', 0
DMZ 1 . "7#1" . 32 . 32 . 32+94 . 32+8 . 128
```