

# Pasmo documentation

(C) 2004-2006 Julián Albo.

Use and distribution allowed under the terms of the GPL license.

Last revision date: 23-jan-2006

Current Pasmo version: 0.6.0 (in progress)

## Index

Introduction

Installation

Command line use

Code generation modes

- Default

- bin

- com

- dump

- hex

- prl

- rel

- cmd

- tap

- tzx

- cdt

- tapbas

- tzxbas

- cdtbas

- plus3dos

- amsdos

- msx

Symbol table

Source code format

Numeric literals

String literals

Identifiers

Labels

Directives

- .8080

- .8086

- .DEPHASE

- .ERROR

- .PHASE

- .SHIFT

- .WARNING

- .Z80

- ASEG

- CSEG

- DEFB

- DEFL

- DEFM

DEFS  
DEFW  
DS  
DSEG  
DW  
ELSE  
END  
ENDIF  
ENDM  
ENDP  
EQU  
EXITM  
IF  
IF1  
IF2  
IFDEF  
IFDEF  
INCLUDE  
INCBIN  
IRP  
IRPC  
LOCAL  
MACRO  
ORG  
PROC  
PUBLIC  
REPT  
SET

## Operators

Table of precedence

List of operators

## Directives

.SHIFT  
ENDM  
EXITM  
IRP  
IRPC  
MACRO  
REPT

## Macros

Suggestions and possible improvements

Tricks

Bugs

Epilogue

# Introduction

Pasmo is a multiplatform Z80 and 8080 cross-assembler, easy to compile and easy to use. It can generate object code in several formats suitable for many Z80 machines and emulators. Pasmo generates fixed position code, can not be used to create relocatable object files for use with linkers.

Pasmo is compatible with the syntax used in several old assemblers, by supporting several styles of numeric and string literals and by providing several names of the most used directives. However, in Pasmo the Z80 mnemonics, register and flags names and directives are reserved words, this may require changes of symbol names conflicting in some programs.

Pasmo can also generate the 8086 equivalent to the z80 assembly code. It can create COM files for MS-DOS, by using the binary generation mode, or CMD files for CP/M 86, by using the --cmd generation mode. This feature is experimental, use with care.

# **Installation**

Download Pasmó from <http://www.arrakis.es/~ninsesabe/pasmo/>.

Several binary executable are provided in the web, if your platform is not between these, or wants a more recent version, you must download the source package and compile it. If you want to compile it in windows you can use cygwin or mingw with the Makefile provided, with other compilers you may need to create a project, workspace or whatever your compiler or IDE uses.

To compile you need gcc version 2.95 or later, with the c++ language included (usually a package called g++-something). Other compilers may also be used, any reasonable standard complaint c++ compiler must compile it with few or none corrections. From version 0.5.2 a configure script is provided. You can use the usual './configure ; make ; make install' procedure. You have also an official Debian package.

# Command line use

Pasmo is invoked from command line as:

```
pasmo [options] [file.asm] [file.bin] [file.symbol [file.publics] ]
```

Where file.asm is the source file, file.bin is the object file to be created and optionally file.symbol is the file where the symbol table will be written and file.publics is the file for the public symbols table. Both symbol file names can be an empty string for no generation or - to write in the standard output. When the --public option is used this is handled in another way, see below. The source and object file can also be specified with the options --input and --output. If the --link option is used the source file must be omitted.

Default mode: If none of the code generation options is specified, then --bin mode is used by default.

Options can be zero or more of the following:

-1 ('one')	Same as --debinfo1
-8	Same as --w8080
-B	Same as --bracket
-E	Same as --equ
-I (UC i)	Same as --include-dir
-S	Same as --skiplines
-d	Same as --debinfo
-h	Same as --help
-m	Same as --module
-o	Same as --output
-v	Same as --verbose
--86	Generate 8086 code instead of Z80. This feature is experimental.
--after	Place common segments after absolute segment when linking.
--alocal	Use autolocal mode. In this mode all labels that begins with '_' are locals. See the chapter about labels for details.
--amsdos	Generate the object file in AMSDOS format.
--asm	Sets the assembly language used at start (can be overridden in the source with the .8080 and .Z80 directives). Possible values are Z80 and 8080. If not specified, Z80 mode is assumed.

--bin	Generate the object file in raw binary format without headers.
--bracket	Use bracket only mode. In this mode the parenthesis are valid only in expressions, for indirections brackets must be used.
--cdt	Generate the object file in .cdt format.
--cdtbas	Same as --cdt but adding a Basic loader before the code.
--cmd	Generate the object file in CP/M 86 CMD format.
--com	Generate the object file in COM format.
--debinfo	Show debug info during second pass of assembly.
--debinfo1	Show debug info during both passes of assembly.
--dump	Generate the object file in readable hex dump format.
--equ	Predefine a symbol. Predefined symbols are treated in a similar way to defineds with EQU. Some possible uses are exemplified in the black.asm example file. The syntax is: '--equ label=value' where label must be a valid label name and value a numeric constant in a format valid in pasmo syntax. The part =value is optional, if not specified the value assigned is FFFF hex.
--err	Direct error messages to standard output instead of error output (except for errors in options).
--help	Show short information about options in standard error and exit.
--hex	Generate the object file in Intel HEX format.
--include-dir	Add directory to the list for searching files in INCLUDE and INCBIN.
--input	Specify the file name to assembly. If this option is specified, the argument file.asm must be omitted.
--link	Link only. Do not assembly any source, just link relocatable modules.
--listing	Specify a file name to generate an assembly listing.
--module	Specify a REL module to link.
--msx	Generate the object file in MSX format.
--name	Name to put in the header in the formats that use it. If unspecified the object file name will be used.
--nocase	Make identifiers case insensitive.
--numerr	Set the numbers of non fatal errors that can be reported before stop the assembly. If not specified a value of 1 is assumed. A value of 0 means to never stop (almost, but PasmO

will probably run out of memory before reaching the limit).

--output	Specify the file name where to put the object code generated. If this option is specified, the argument file.bin must be omitted.
--plus3dos	Generate the object file in PLUS3DOS format.
--prl	Generate the object file in CP/M PRL format.
--public	Only the public symbols table is generated, using the file.symbol name, file.symbol must not be specified when using this option.
--skiplines	Skip the number of lines specified at the begin of the input file. Useful for debugging during PasmO development, don't know if it can have other interesting uses.
--tap	Generate the object file in .tap format.
--tapbas	Same as --tap but adding a Basic loader before the code.
--txx	Generate the object file in .txx format.
--txbas	Same as --txx but adding a Basic loader before the code.
--verbose	Verbose mode. Show progress information about loading of files, progress of assembly and maybe other things.
--version	Show PasmO version information in standard output and exit.
--w8080	Show warnings when Z80 instructions that have no equivalent in 8080 are used. Makes easy to write programs for 8080 processor using Z80 assembler syntax.

The -d option is intended to debug pasmo itself but can also be useful to find errors in asm code. When used the information is shown in the standard output. Error messages go to error output unless the --err option is used.

# Code generation modes

Default mode: If none of the code generation options is specified, then --bin mode is used by default.

- bin        This mode just dumps the code generated from the first position used without any header. This mode can be used for direct generation of CP/M or MSX COM files, supposed that you use a ORG 100H directive at the beginning of the code, or to generate blocks of code to be INCBINed in other programs.
- com        This mode is similar to --bin, but code is assembled or linked starting at 0100 hex without the need to ORG it. The reason for the use of the two options is to make it easier to assemble without changes code written for DR ASM or similar assemblers and code written for use with a relocatable assembler such as DR RMAC and a linker.
- dump       This mode generates a human readable uppercase hexadecimal dump, consisting of lines of 16 hexadecimal two digit numbers separated by spaces with the address of the code at the beginning of the line as a four digit hexadecimal number followed by a colon and a space.
- hex        This mode generates code in Intel HEX format. This format can be used with the LOAD or HEXCOM CP/M utilities, can be transmitted more easily than a binary format, and is also used in some PROM programmers.
- prl        The prl format is used in several variants of Digital Research CP/M operating system. In pasmo it is supported only to create RSX files for use in CP/M Plus, use for PRL files in MP/M is not supported because I don't have a MP/M system, real or emulated, where to test it.
- rel        The REL relocatable format is used by Digital Research and Microsoft assemblers and compilers, and many others. This version of Pasmo has preliminary support for it, supporting only program segment, PUBLIC symbols, and sources that not use ORG directives. Note that for compatibility you probably must use the --nocase mode, or write all public symbols in upper case. Remember also that in REL files the identifiers are truncated to 6 characters.
- cmd        This option generates a CP/M 86 CMD mode, using the 8080 memory model of CP/M 86. Used in conjunction with the --86 option can easily generate CP/M 86 executables from CP/M 80 sources with minimal changes.
- tap        The --tap options generates a tap file with a code block, with the loading position set to the beginning of the code so you can load it from Basic with a LOAD "" CODE instruction.
- ttx        Same as --tap but using ttx format instead of tap.
- cdt        This option generates a cdt file with a code block, with the loading position set to the beginning of the code and the start address to the start point specified in the source, if any, so you can use RUN "" to execute it or LOAD "" to load it.
- tapbas     With this option a tap file is generated with two parts: a Basic loader and a code block with the object code. The Basic loader does a CLEAR before the initial address of



the code, loads the code, and executes it if a entry point is defined (see the END directive). That way you can directly launch the code in a emulator, or transfer it to a tape for use in a real Spectrum.

- tzxbas      Same as --tapbas but using tzx format instead of tap.
- cdtbas      Same as --tapbas but using cdt format instead of tap and with a Locomotive Basic loader instead of Spectrum Basic.
- plus3dos    Generate the object file in plus3dos format, used by the Spectrum +3 disks. The file can be loaded from Basic with a LOAD "filename" CODE instruction.
- amsdos      Generate the object file with Amsdos header, used by the Amstrad CPC on disk files. The file generated can be loaded from Basic with LOAD "filename", address or executed with RUN "filename" if an entry point has been specified in the source (see the END directive).
- msx          Generate the object file with header for use with BLOAD in MSX Basic.

# **Symbol table**

The symbol table generated contains all identifiers used in the program, with the locals represented as a 8 digit hexadecimal number in order of use, unless the --public option is used. In that case only the symbols specified in PUBLIC directives are listed.

The symbol table format is a list of EQU directives. That way you can INCLUDE it in another source to create programs composed of several blocks.

# Source code format

Source code files must be valid text files in the platform used. The use of, for example, Unix text files under pasmo in windows, is unsupported and the result is undefined (may depend of the compiler used to build pasmo, for example). The result of the use of a file that contains vertical tab or form feed characters is also undefined.

Some symbols have several meanings depending on its use and the context, this is caused by the intent to be source compatible with several old assemblers and to allow the use of operators commonly used in another languages. The recommended way to avoid mistakes is to always separate the operators and its operands with white space, especially inside macros.

Everything after a ; in a line is a comment (unless the ; is part of a string literal, of course). There are no multiline comments, you can use IF 0 .... ENDIF instead (but see INCLUDE). If the comment begins with ;; instead of a single ; it will not be included in macro expansion.

String literals are written to the object file without any character set translation. Then the use of any character with a different meaning in the platform where pasmo is running and the destination machine must be avoided, and the code of the character may be used instead. That also means that using Pasmo in any machine that uses a non-ascii compatible character set may be difficult, and that a source written in utf-8 may give undesired results. This may be changed in future versions of Pasmo.

A line may begin with a decimal number followed by blanks. This number is ignored by the assembler, is allowed for compatibility with old assemblers. The line number reported in errors is the sequential number of the line in the file, not this.

Blanks are significative only in string literals and when they separate lexical elements. Any number of blanks has the same meaning as one. A blank between operators and operands is allowed but no required except when the same character has other meaning as prefix ('\$' and '%', for example).  
Literals.

# Numeric literals

Numeric literals can be written in decimal, binary, octal and hexadecimal formats. Several formats are accepted to obtain compatibility with the source format of several assemblers.

&Hnnnn	Hexadecimal constant, digits from 0 to 9 and A to F
0xnnnn	
\$nnnn	Except for the lone \$ operator (See operators section)
#nnnn	Except for the ## operator (See operators section)
&Onnnn	Octal constant, digits from 0 to 7
%nnnn	Binary constant, digits from 0 to 1

A literal that begins with & can be hexadecimal, octal or binary constant, depending on the character that follows the &: H means hexadecimal, O octal and X hexadecimal, if none of this the character must be a valid hexadecimal digit and the constant is hexadecimal. See also the use of & in macros.

A literal that begins with % is a binary constant. See also the use of % in macro arguments.

A literal that begins with a decimal digit can be a decimal, binary, octal or hexadecimal. If the digit is 0 and the following character is an X, the number is hexadecimal. If not, the suffix of the literal is examined: D means decimal, B binary, H hexadecimal and O or Q octal, in any other case is taken as decimal. Take care, FFFFh for example is not an hexadecimal constant, is an identifier, to write it with the suffix notation you must do it as 0FFFFh.

All numeric formats can have \$ signs between the digits to improve readability. They are ignored.

# String literals

There are two formats of string literals: single or double quote delimited.

A string literal delimited with single quotes is the simpler format, all characters are included in the string without any special interpretation, with the only exception that two consecutive single quotes are taken as one single quote character to be included in the string. For example: the single quote delimited string 'That"s all folks' generates the same string as the double quote delimited "That's all folks".

A string literal delimited with double quotes is interpreted in a way similar to the C and C++ languages. The \ character is taken as escape character, with the following interpretations: n is a new line character (0A hex), r is a carriage return (0D hex), t is a tabulator (09 hex), a is a bell (07 hex), x indicates that the two next characters will be considered the hexadecimal code of a char and a char with that code is inserted, an octal digit prefixes and begins an octal number of up to three digits, and the corresponding character is inserted into the string, the characters \ and " means to insert itself in the string, and any other char is reserved for future use.

A string literal of length 1 or 2 can be used as a numeric constant with the numeric value of the first character, and the second in this case as the high order byte. This allows expressions such as 'A' + 80h to be evaluated as expected.

# Identifiers

Identifiers are the names used for labels, EQU and DEFL symbols and macro names and parameters. The names of the Z80 mnemonics, registers and flag names, and of pasmo operands and assemble directives are reserved and can not be used as names of identifiers, except in macro parameters. Reserved names are case insensitive, even if case sensitive mode is used.

In the following 'letter' means an English letter character in upper or lower case. Characters that correspond to letters in other languages are not allowed in identifiers.

Identifiers begin with a letter, '\_', '?', '@' or '.', followed by zero or more letters, decimal digit, '\_', '?', '@', '.' or '\$'. The '\$' are ignored, but a reserved word with a '\$' embedded or appended is not recognized as such.

Identifiers that begin with '\_' are special when using autolocal mode, see the --alocal option and the chapter about labels for details. The check for autolocal is done before stripping '\$', then \$\_name is not considered local.

Identifiers are case sensitive if the option --nocase is not used. When using --nocase, they are always converted to upper case.  
File names.

File names are used in the INCLUDE and INCBIN directives. They follow special rules.

A file name that begins with a double quote character must end with another double quote, and the file name contains all character between them without any special interpretation.

A file name that begins with a single quote character must end with another single quote, and the file name contains all character between them without any special interpretation.

In any other case all characters until the next blank or the end of line are considered part of the file name. Blank characters are space and tab.

# Labels

A label can be placed at the beginning of any line, before any assembler mnemonic or directive. Optionally can be followed by a ':', but is not recommended to use it in directives, for compatibility with other assemblers, however it may be needed if a macro with the same name as the label is already defined.

A line that has a label with no mnemonic nor directive is also valid.

The label has special meaning in the MACRO, EQU and DEFL directives, in any other case the value of the current code generation position is assigned to the label.

Labels can be used before its definition, but the result of doing this with labels assigned with DEFL is undefined.

The value of a label cannot be changed unless DEFL is used in all assignments of that label. If the value assigned to a label is different in the two passes of the assembly the program is illegal, but is not guaranteed that an error is generated. However, is legal to assign a value undefined in the first pass (by using an expression that contains a label not yet defined, for example).

In the default mode a label is global unless declared as LOCAL into a MACRO, REPT or IRP block, see the LOCAL directive for details.

In the autolocal mode, introduced by using the --alocal command line option, all labels that begin with a '\_' are locals. Its ambit ends at the next non local label or in the next PROC, LOCAL, MACRO, REPT, IRP, IRPC, ENDP or ENDM directive. The check for autolocals is done before stripping the '\$' in the identifier, thus `$_this_label_is_not_autolocal`.

Both automatic and explicit local labels are represented in the symbol table listing as 8 digit hexadecimal numbers, corresponding to the first use of the label in the source.

# Directives

List of directives supported in Pasm0, in alphabetical order.

.8080	Sets the 8080 assembly language mode. After executing this directive and until the end of file or a .Z80 directive, the source is interpreted as 8080 assembly code.
.8086	Reserved for future use.
.DEPHASE	Terminates the effect of .PHASE, after this the code is generated relative to the current position counter.
.ERROR	Generates an error during assembly if the line is actively used, that is, in a macro if it gets expanded, in an IF if the current branch is taken. All text following the directive is used as error message.
.PHASE	Take a numeric expression as argument. The code generated after this and until a .DEPHASE or the end of the program is generated relative to the position specified in the argument instead of the current position.
.SHIFT	Shift MACRO arguments, see the chapter about macros.
.WARNING	Same as .ERROR but emitting a warning message instead of generating an error.
.Z80	Sets the Z80 assembly language mode. After executing this directive and until the end of file or a .8080 directive, the source is interpreted as Z80 assembly code.
ASEG	Absolute segment. Code generated until other segment directive is generated in absolute address mode.
CSEG	Code segment. Code generated until other segment directive is generated in code segment relative address mode.
DB	Define Byte. The argument is a comma separated list of string literals or numeric expressions. The string literals are inserted in the object code, and the result of the numeric expression is inserted as a single byte, truncating it if needed.
DEFB	DEFine Byte, same as DB.
DEFL	DEFine Label. Must be preceded by a label. The argument must be a numeric Expression, the result is assigned to the label. The label used can be redefined with other DEFL directive. DEFL is not recognized in 8080 mode, use SET instead.
DEFM	DEFine Message, same as DB.
DEFS	DEFine Space, same as DS.
DEFW	Same as DW.
DS	Define Space. Take one or two comma separated arguments. The first or only argument is the amount of space to define, in bytes. The second is the value used to fill the space,



if absent 0 will be used.

DSEG	Data segment. Code generated until other segment directive is generated in data segment relative address mode.
DW	Define Word. The argument is a comma separated list of numeric expressions. Each numeric expression is evaluated as a two byte word and the result inserted in the code in the Z80 word format.
ELSE	See IF
END	Ends the assembly. All lines after this directive are ignored. If it has an argument it is evaluated as a numeric expression and the result is set as the program entry point. The result of setting an entry point depends of the type of code generation used, may be none but even in this case may be used for documentation purposes.
ENDIF	See IF
ENDM	Ends a macro, see the chapter about macros.
ENDP	Marks the end of a PROC block, see PROC.
EQU	EQUate. Must be preceded by a label. The argument must be a numeric expression, the result is assigned to the label. The label used can't be redefined.
EXITM	Exits a macro, see the chapter about macros.
IF	Conditional assembly. The argument must be a numeric expression, a result of 0 is considered as false, any other as true. If the argument is true the following code is assembled until the end of the IF section or an ELSE directive is encountered, else is ignored. If the ELSE directive is present the following code is ignored if the argument was true, or is assembled if was false. The IF section is ended with a ENDIF or a ENDM directive (in the last case the ENDM has also its usual effect). IF can be nested, in that case each ELSE and ENDIF takes effect only on its corresponding IF, but ENDM ends all pending IF sections.
IF1	Conditional assembly during first pass.
IF2	Conditional assembly during second pass.
IFDEF	Conditional assembly if the argument, an identifier, is defined.
IFNDEF	Conditional assembly if the argument, an identifier, is not defined.
INCLUDE	Include a file. See the file names chapter for the conventions used in the argument. The file is read and the result is the same as if the file were copied in the current file instead of the INCLUDE line. The file included may contain INCLUDE directives, and so on. The same file can be included several times, the file is read just one time, provided the complete path and name is written equally. INCLUDE directives are processed before the assembly phases. Because of this, before version 0.6.0 the use of IF directives to conditionally include different files were not allowed. Now the file not

opened result is stored and only generates an error if the line is actively assembled. The result of using unpaired directives in an included file is undefined, do it at your own risk.

INCBIN	INClude BINary. Include a binary file. Reads a binary file and insert its content in the generated code at the current position. See the file names chapter for the conventions used in the argument.
IRP	Repeat a block of code substituting arguments. See the chapter about macros.
IRPC	Repeat a block of code for each char in the argument. See the chapter about macros.
LOCAL	Marks identifiers as local to the current block. The block may be a MACRO, REPT, IRP or PROC directive, the local ambit ends in the corresponding ENDM or ENDP directive. The ambit begins at the LOCAL directive, not at the beginning of the block, take care with that. If several LOCAL declarations of the same identifier are used in the same block, only the first has effect, the others are ignored.
MACRO	Defines a macro, see the chapter about macros.
ORG	ORiGin. Establishes the origin position where to place generated code. Several ORG directives can be used in the same program, but if the result is that code generate overwrites previous, the result is undefined.
PROC	Marks the begin of a PROC block. The only effect is to define an ambit for LOCAL directives. The block ends with a corresponding ENDP directive. The recommended use is to enclose a subroutine in a PROC block, but can also be used in any other situation.
PUBLIC	The argument is a comma separated list of identifiers. Each identifier is marked as public. When using the --public command line option only the identifiers marked as public are included in the symbol table listing.
REPT	REPeaTs a block. See the chapter about macros.
SET	Only available in 8080 mode. Same as DEFL in Z80 mode.

# Operators

All numeric values are taken as 16 bits unsigned, using 2 complement or truncating when required. Logical operators return FFFF hex for true and 0 for false, in the arguments 0 is false and any other value true.

Parenthesis may be used to group parts of expressions. They are also used to express indirections in the z80 instructions that allows or require it. This can cause some errors when a parenthesized expression is used in a place where an indirection is allowed. Pasmu uses some heuristic to allow the expression to be correctly interpreted, but are far from perfect.

Using the bracket only mode the parenthesis have the unique meaning of grouping expressions, brackets are required for indirections, thus solving ambiguities.

Short circuit evaluation: the && and || operators and the conditional expression are short circuited. This means that if one of its operators need not be evaluated, it can include undefined symbols or divisions by 0 without generating an error (but still must have correct syntax). In the conditional expression this applies to the branch not taken, in the && operator to the second operand if the first is false, and in the || operator to the second operand if the first is true.

Table of precedence.

Table of operators by order of precedence, those in the same line have the same precedence:

```
## (see note)
$, NUL, DEFINED
*, /, MOD, %, SHL, SHR, <<, >>
+, - (binary)
EQ, NE, LT, LE, GT, GE, =, !=, <, >, <=, >=
NOT, ~, !, +, - (unary)
AND, &
OR, |, XOR
&&
||
HIGH, LOW
?
```

The ## operator is a special case, is processed during the macro expansion, see the chapter about macros.

Note that the precedence is not the same as in some old assemblers, especially MASM. Always mark precedence with parenthesis in code intended to be use with several different assemblers.

Note also that HIGH and LOW are operators, not functions. To avoid confusion with precedence rules the syntax required is not HIGH (argument), but (HIGH argument), or even (HIGH (argument) ) inside macros if the argument contains macro parameters.

Alphabetic list of operators.

! Logical not. Returns true if its argument is 0, false otherwise.

!= Same as NE.

##	Identifier pasting operator, see the chapter about macros.
\$	Gives the value of the position counter at the beginning of the current sentence. For example, in a DW directive it gives the position of the first item in the list, not the current item.
%	Same as MOD
&	Same as AND
&&	Logical and. True if both operands are true
*	Multiplication
+	Addition or unary +
-	Subtraction or unary -
/	Integer division, truncated
<	Same as LT
<<	Same as SHL
<=	Same as LE
=	Same as EQ
>	Same as GT
>=	Same as GE
>>	Same as SHR
?	Conditional expression. Must be followed by two expressions separated by a :, if the expression on the right of ? is true, the first expression is evaluated, if false, the second.
	Same as OR
	Logical or. True if one of the operands is true
~	Same as NOT
AND	Bitwise and operator
DEFINED	The argument must be an identifier. The result is true if the identifier is defined, false otherwise.
EQ	Equals. True if both operands are equal, false otherwise.
GE	Greater than or equal to. True if the left operand is greater or equal than the right.

GT	Greater than. True if the left operand is greater than the right.
HIGH	Returns the high order byte of the argument.
LE	Less than or equal to. True if the left operand is lesser or equal than the right.
LOW	Returns the low order byte of the argument.
LT	Less than. True if the left operand is lesser than the right.
MOD	Modulus. The remainder of the integer division.
NE	Not equal. False if both operands are equal, true otherwise.
NOT	Bitwise not. Return the ones complement of its operand.
NUL	Returns true if there is something at the right, else returns false. Useful if the Arguments are parameters of macros.
OR	Bitwise or operator
SHL	Shift left. Returns the left operand shifted to the left the number of bits specified in the Right operand, filling with zeroes.
SHR	Shift right. Returns the left operand shifted to the right the number of bits specified in the right operand, filling with zeroes.
XOR	Bitwise XOR (exclusive or) operator

# **Macros**

There are two types of macro directives: the proper MACRO directive and the repetition directives REPT and IRP. In addition the ENDM and EXITM directives controls the end of the macro expansion.

Parameters.

A macro parameter is an identifier that when the macro is expanded is substituted by the value of the argument applied. The identifier used can have the same name of a keyword, the keyword is not recognized as such in that case. Be careful, the readers of the macro code may get confused with that.

# Parameter expansion

By default, the parameters inside of a macro are expanded by substituting it with the corresponding argument in the macro call.

If a MACRO is defined inside another macro directive the external parameters are not substituted, with the other macro directives the parameter substitution is done beginning by the most external directive.

The NUL operator can be used to check if the argument passed to the parameter is not empty. The .SHIFT directive can be used to work with an undetermined number of arguments.

Identifier pasting: inside a macro the operator `##` can be used to join two identifiers resulting in another identifier. This is intended to allow the creation of identifiers dependent of macro arguments.

Forced expansion: the `&` is used to explicitly mark the expansion of a macro argument following it without whitespace. It also allows parameter expansion inside a string literal. Unlike the `##` operator, when using it to create identifiers it does not suppress whitespace preceding it.

# Arguments

Macro arguments are passed literally by default. This is not desirable in cases when the value of an expression, because precedence rules can modify the result of the expansion inside other expressions. In those cases, the % operator can be used at the beginning of a macro argument, it evaluates the expression following, takes its value as a numeric literal in decimal and passes it as the effective macro argument.

There are two types of macro arguments: the first is a comma separated list of items, where each item is an arbitrary list of tokens. The second is a comma separated list of items between angle brackets. Each item can be any token, or another angle bracket delimited expression. whitespace between items is ignored.

The nested angle bracket delimited arguments can be used to include in the argument whitespace or commas, or to pass arguments to be used as arguments of another macro. Calling a macro.

A macro defined with the MACRO directive is called by using its name as if it were a directive or instruction. The macro arguments are evaluated and assigned to the MACRO parameters. If there are less arguments than parameters, the remainder get assigned an empty value. If there are more, the remaining arguments are stored but not assigned to any parameter; they can only be accessed by using the .SHIFT directive inside the macro.

There are some special cases when using the name of an already defined macro name:

This is a label with the same name as the macro:

```
macroname: ....
```

This is a redefinition of macroname:

```
macroname MACRO ...
```

In the following cases MACRO is taken as the beginning of the arguments of macroname. I don't know any possible good use of this so it may be banned in future versions. If you consider it useful, please send me code samples of its use.

```
otherlabel macroname MACRO ...  
otherlabel: macroname MACRO ...
```



# Directives

.SHIFT	Can be used only inside a MACRO. The MACRO arguments are shifted one place to the left, the first argument is discarded. If there are not enough arguments to fill the parameter list after the shift, the remaining arguments get undefined.
ENDM	Marks the end of the current MACRO definition, or the current REPT or IRP block. All IF blocks contained in the macro block are also closed.
EXITM	Exits the current MACRO, REPT or IRP block. In the case of MACRO, the macro expansion is finished, in the other cases the code generation of the block is terminated and the assembly continues after the corresponding ENDM.
IRP	IRP parameter, argument list. Repeats the block of code between the IRP directive and its corresponding ENDM one time for each of the arguments.
IRPC	IRPC parameter, character list. Repeats the block of code between the IRPC directive and its corresponding ENDM one time for each character in the list. The character list can be a literal string or a macro argument between angle brackets.
MACRO	<p>Defines a macro. There are two forms that can be used:</p> <p style="padding-left: 40px;"><i>name MACRO [ list of parameters]</i></p> <p style="padding-left: 80px;">or:</p> <p style="padding-left: 40px;"><i>MACRO name [ , list of parameters]</i></p> <p>In all cases, list of parameters is a comma separated list of identifiers, and name is the name assigned to the macro created. A macro is used by simply specifying its name, and optionally a list of arguments. The arguments list does not need to have the same length as the parameter list of the macro. If it is longer, the extra arguments are not used, but can be retrieved by using .SHIFT inside the MACRO. If it is shorter some parameters get undefined, this can be tested inside the MACRO by using the NUL operator.</p>
REPT	<p>Repeats the block of code between the REPT directive and its corresponding ENDM the number of times specified in its argument. The argument can be 0, in that case the block is skipped. Additionally, a loop var can be specified. This var is not a macro parameter, is used as a LOCAL DEFL symbol, whose value is incremented in every loop iteration. The initial value and increment can be specified, with defaults of 0 and 1 respectively.</p> <p style="padding-left: 40px;">REPT count</p> <p style="padding-left: 40px;">REPT count, varname</p> <p style="padding-left: 40px;">REPT count, varname, initial</p> <p style="padding-left: 40px;">REPT count, varname, initial, increment</p> <p>Update: In 0.6.0 the loop var is no longer a symbol, it is expanded like a macro argument. That way will be much more useful with the new macro expansion capabilities.</p>

# Suggestions and possible improvements

The assumption of Pasmio is that, being a cross-assembler, it will be used on a machine with many available resources. Then I do not make any effort to provide means to do things that can be easily made with other utilities, unless I think (or other people convince me) that including it in Pasmio can be much more convenient.

For example, if you want to create a sin table you can write a program in your favorite language that writes a file with the table and INCLUDE that file, and if you want to automate that type of things you can use make.

Taking that into account, I am open to suggestions to improve Pasmio and to patches that implement it. In the latter case please take care to write things in a portable way, without operating system or compiler dependences.

Note: Please do not send patches during the current development of 0.6.0 version, I'm still rewriting and rearranging some parts and integrating patches will be difficult. E-mail me about the feature desired or bug, instead.

Why can't Pasmio generate linkable code? Pasmio has a simple code generator that uses absolute address of memory. That will make it difficult to adapt it to generate relocatable code for use with linkers. I don't have plans to do it for the moment, maybe someone wants to contribute?

Update: Starting with version 0.6.0 pasmio can generate linkable code in REL format, and link modules in that format. This feature is currently unfinished and poorly tested, please use with care and report bugs.

Game Boy? Some people suggested to add support for Game Boy programming. There are two problems, the simplified way used to generate code in Pasmio, and my non-existent knowledge of the Game Boy. Thanks.

Thanks to all people that have made suggestions and notified me of or corrected bugs. And to these that show me the beautiful things they do with Pasmio.

# Tricks

You can use PasmO to convert any binary file to .tap, just write a tiny program called for example convert.asm:

```
ORG address_to_load_the_file
INCBIN file.bin
```

Assemble it with: `pasmO --tap convert.asm file.tap`, and you have it. The same may be done for the other formats supported.

To obtain the code of an instruction you can do: `echo 'ld a,b' | pasmo --input - -o - --dump`

# Bugs

PasmO emits a warning when using a expression that looks like a non-existent z80 instruction, such as 'ld b, (nn)', but the simplified way used to detect that also warns in cases like: 'ld b,(i1+i2)\*(i3+i4)'. A way to avoid the warning in that case is to prefix the expression with parenthesis with '+' or '0 +'. Using the bracket only mode the problem does not exist, in that case the parenthesis are always taken as expressions (and the programmer is supposed to know that), thus the warning is not emitted. More suggestions about that are welcome.

Update: The new parser in 0.6.0 does a much better work, warning only if the expression is entirely inside parenthesis.

There is no way to include a file whose name contains blanks, single and double quotes. Does someone use file names like that?

# Epilogue

That's all folks!

Send comments and criticisms to:  
julian.notfound@gmail.com