zasm - Z80 Assembler – Version 4.4

User Manual from the web documentation from last update: 2022-04-20 17:31:36

Crawled by XianBild on 30.05.2025

Content

Quick Overview

Features

zasm is an assembler for the Z80 CPU.

It can also assemble code for the extended instruction set of the Z180 / Hitachi HD64180, as well as code limited to the Intel 8080 e.g. for CP/M. **zasm** can also assemble native 8080 assembler source. By the help of sdcc zasm can include c source files. A trimmed version of sdcc is available from the download page. **zasm** can create binary, Intel Hex or Motorola S19 files and various special files for Sinclair and Jupiter Ace emulators:

ZX Spectrum: .sna .z80 .tap

Jupiter Ace: .ace .z80[1] .tap

ZX80: .o / .80 .z80[1]

ZX81: .p / .81 / .p81 .z80[1]

zasm can translate the character set for the target system, e.g. for the ZX80 and ZX81.

The list file can include accumulated cpu cycles. Various historically used syntax variants and the syntax emitted by sdcc are supported. Multiple code and data segments can be defined, nested conditional assembly and nested local scopes are available.

Last not least, **zasm** supports macros.

[1]) eventually only supported by zxsp.

Typical invocation

Compile the file "emuf_rom.asm" and store a plain list file without generated opcodes and the output file in the same folder. The source file may include other files, either binary, assembler sources or c files:

```
zasm emuf_rom.asm
```

Include generated opcodes, accumulated cpu cycles and a labels listing in the list file:

```
zasm -uwy emuf_rom.asm
```

Assemble source for the Intel 8080 assembler:

```
zasm --asm8080 emuf8080_rom.asm
```

Source file examples

The source file must be either 7-bit ascii or utf-8 encoded. If you still use a legacy encoding for your text files then this is OK if non-ascii characters occure only in comments. You just won't be able to put non-ascii characters in strings or character literals.

Line breaks in the source file may use any well known format, but the text files generated by zasm will use newline (character 0x0A) only.

Basic source file which does not set a target. Instruction 'org' is required:

```
; my little EMUF

; define some labels:
anton   equ 66
berta   equ 88

; define code:
        org   0
reset:  di
        jp    _init

        org   8
printc: jp    _printc

        ...

; pad file to eprom size:

ds  0x2000 - $
```

Basic source file which sets a target and uses code and data segments:

```
; my little EMUF

#target rom

; define a data segment in ram and therein some variables:

#data _SYSVARS,0x4000

anton   data    2
berta   data    4
caesar  data    1

; define a code segment for an eprom:

#code _EPROM,0,0x4000

reset:  di
        jp    _init

        org   8
printc: jp    _printc

        ...
```

Differences from v3 to v4

Command line options

The command line options have changed: start `zasm` with no options to see a summary.

See chapter 'Command line options' for details.

Include C sources

zasm 4.0 can include c source files, with the help of `sdcc`. A trimmed version of sdcc for OSX can be found on the zasm download page.

C source files are #included just like assembler source files. They are compiled using sdcc and the output file is #included into the total source.

After the last included files you can use '
#include library' to resolve undefined

labels from the system library or any other library directory.

See chapter 'Including C source files' for details.

Assemble 8080 assembler source

zasm 4.0 can assemble native 8080 assembler source files if it is invoced with command line option '--asm8080'.

See chapter '8080 Assembler' for details.

#code and #data

The syntax for `#code` and `#data` has changed. To support the c compiler an additional first argument, the segment name, is now required and multiple code and data segments are allowed. E.g.:

```
	#code _NAME, _start, _size
```

It is possible to switch to any defined segment at any time using #code, #data or .area with the name as a single argument.

Pseudo instruction 'data' is still available but must only be used after the source explicitely switched to a data segment.

For simple projects it is still possible to use the traditional style with just defining an `org` at the start of the source file, which was possible with recent versions of v3, but v4 no longer complains about it.

See chapter #target, #code and #data for details.

New handling of 'org'

Instead of using #target and #code, you can use `org` in the traditional way to set the origin of your code. This is done by the first 'org' instruction in your source which must preceed any code generating instruction. (As it was also possible with recent versions of v3.)

The behavior of `org` inside code has changed: in v4 it does insert space up to the requested new address. For the old behavior to just set a new logical code origin use the new pseudo instructions `.phase` and `.dephase`.

See chapter 'org' for details.

Source layout for various targets

The required layout of #code segments for some targets has changed and some targets are supported in v4 for the first time.

Supported targets in v4:

```
	BIN, ROM, SNA, Z80, TAP (ZX Spectrum and Jupiter Ace), O/80, P/81/P81, ACE, TZX
```

See chapter 'Targets' for details.

Macros and rept

Macros and `rept` are now supported. zasm also supports 'defl' and 'set' for redefinable labels and conditional code ranges in macros with 'if' and 'endif'. Arguments which break normal syntax rules can be passed with '<' … '>'.

See chapter 'macro' for details.

Misc. other features

The list file can include accumulated cpu cycles: see command line option --cycles.

#local … #endlocal for nested local scopes.

#charset for automatic conversion of strings and character literals into the target's character set.

Limit instructions and registers to the 8080 with command line option --8080.

Enable additional instructions of the Z180 / HD64180.

Pseudo instructions can be written in many variants for compiling old sources without modifications.

Many convenience 'compound' opcodes like "ld a,(hl++)" or "ld bc,de"

Function 'opcode(opcodename)' to get the major byte of an opcode.

Multiple opcodes per line separated with '\'.

Command line options

When invoking zasm you can append options to it on the command line, most notably the name of the source file.

```
zasm [options] [-i] inputfile [[-l] listfile|dir] [[-o] outfile|dir]
```

default output dir = source dir

default list dir = output dir

Typical invocations

zasm -uwy emuf_rom.asm

List of command line options as printed by zasm:

```
-u  --opcodes   include object code in list file
-w  --labels    append label listing to list file
-y  --cycles    include cpu clock cycles in list file
-b  --bin       write output to binary file (default)
-x  --hex       write output in intel hex format
-s  --s19       write output in motorola s-record format
-z  --clean     clear intermediate files, e.g. compiled c files
-e  --compare   compare own output to existing output file
-T  --test      run self test (requires developer source tree)
-g  --cgi       prevent access to files outside the source dir
--maxerrors=NN  set maximum for reported errors (default=30, max=999)
--date=DATETIME for reproducible __date__ and __time__ (default=now)
--target=ram    default to 'ram' not 'rom' => cpu addresses in hex files
-o0             don't write output file
-l0             don't write list file
--z80           target Zilog Z80 (default except if --asm8080)
--z180          enable Z180 / HD64180 instructions
--8080          restrict to Intel 8080 (default if --asm8080)
--asm8080       use 8080 assembler syntax
--convert8080   convert source from 8080 to Z80 mnemonics
-v0 -v1 -v2     verbosity of messages to stderr (0=off, 1=default, 2=more)
--ixcbr2 | …xh  enable ill. instructions like 'set b,(ix+d),r' or 'set b,xh'
--dotnames      allow label names starting with a dot '.'
--reqcolon      colon ':' after program label definitions required
                => label definitions and instructions may start in any
                   column
--casefold      label names are case insensitive (implied if --asm8080)
--flatops       no operator precedence: evaluate strictly from left to
                right
-c path/to/cc   set path to c compiler (default: sdcc in $PATH)
-t path/to/dir  set path to temp dir for c compiler (default: output dir)
-I path/to/dir  set path to c system header dir (default: sdcc default)
-L path/to/dir  set path to standard library dir (default: none)
```

Options can be packed, e.g. '-uwy' is the same as giving 3 separate options.

Also, due to the way zasm parses it's arguments, you can even include options which require an additional argument in this way:

```
$> zasm -cuwy /bin/sdcc emuf.asm
```

Here the 'c' will eat the '/bin/sdcc'.

`-u --opcodes`

`Request` a list file which includes the generated bytes.

```
0215:           ;/pub/Develop/Projects/zasm-4.0/Test/main.c:11: int mult(int a, int
b)
```

```
0215:           ; --------------------------------
0215:           ; Function mult
0215:           ; --------------------------------
0215:           _mult:
0215: DDE5          push    ix
0217: DD210000      ld      ix,#0
021B: DD39          add     ix,sp
021D:           ;/pub/Develop/Projects/zasm-4.0/Test/main.c:13: return a * b;
021D: DD6E06        ld      l,6 (ix)
0220: DD6607        ld      h,7 (ix)
0223: E5            push    hl
0224: DD6E04        ld      l,4 (ix)
0227: DD6605        ld      h,5 (ix)
022A: E5            push    hl
022B: CDA003        call    __mulint
022E: F1            pop     af
022F: F1            pop     af
0230: DDE1          pop     ix
0232: C9            ret
```

-w --labels

Request a labels summary at end of the list file. Also lists the defined code and data segments.

```
; +++ segments +++

#CODE PROG_HEADER:  start=0     len=17    flag=0
#CODE PROG_DATA:    start=0     len=55    flag=255
...
#CODE _CABS:        start=36246 len=0
#CODE _GSFINAL:     start=36246 len=0
#CODE _INITIALIZER: start=36246 len=12
#DATA _DATA:        start=23296 len=53
#DATA _INITIALIZED: start=23349 len=203
; +++ global symbols +++
CODE_DATA           = $5DC0 =  24000  CODE_DATA    test-tap.asm:107 (unused)
CODE_HEADER         = $0000 =      0  CODE_HEADER  test-tap.asm:95 (unused)
_CABS               = $8D96 =  36246  _CABS        test-tap.asm:120 (unused)
_CODE               = $717E =  29054  _CODE        test-tap.asm:119 (unused)
_DABS               = $5C00 =  23552  _DABS        test-tap.asm:137 (unused)
_DATA               = $5B00 =  23296  _DATA        test-tap.asm:135
_GSFINAL            = $8D96 =  36246  _GSFINAL     test-tap.asm:121 (unused)
…
```

-y --cycles

Requests to include accumulated cpu cycles in the list file.

'accumulated' means that zasm does not print the individual execution time for each instruction but sums it up for a thread of code. Program labels are supposed to be entry points from arbitrary positions in code and therefore the sum is reset at every program label. Instructions with varying execution time are given with their run-through time in first place and the branching or repeating time as a second value.

```
                        ; initialize initialized data:
5DC0: 010C00   [10]            ld  bc,_INITIALIZER_len ; length of segment
_INITIALIZER
5DC3: 11355B   [20]            ld  de,_INITIALIZED     ; start of segment
_INITIALIZED
5DC6: 21968D   [30]            ld  hl,_INITIALIZER     ; start of segment
_INITIALIZER
5DC9: 78       [34]            ld  a,b
5DCA: B1       [38]            or  c
5DCB: 2802     [45|50]         jr  z,$+4
5DCD: EDB0     [61|21]         ldir
```

-b --bin

Request to write the output file in binary format. This is the default.

-x --hex

Request to write the output file in Intel Hex format. This is only allowed for target ROM (the default) or target BIN.

```
:200000003E00ED79ED79ED793E40ED793EDAED793E37ED79ED780F30FB3A1027D370ED787B
:090020000F0F30FADB70321027DB
:00000001FF
```

-s --s19

Request to write the output file in Motorola S-Record format. This is only allowed for target ROM (the default) or target BIN.

This format supports a header and zasm uses it to store the source file name and assembly date. Since it is encoded as hex data, it is not directly readable in the example below:

```
S00F000073203230313520D30312D313178
S12C00003E00ED79ED79ED793E40ED793EDAED793E37ED79ED780F30FB3A1027D370ED780F0F30FADB7
032102772
S5030001FB
S9030000FC
```

--target=ram

[4.4.6] If you assemble an old source which uses no #target directive then the target is 'rom' by default. This makes a tiny difference for the fillbyte ($FF instead of $00) and a big difference if you write the output to a hex file which include addresses: target 'rom' stores addresses starting from $0000 for an eprom burner, while target 'ram' stores addresses as set in the #code directives or by the '.org' pseudo opcode suitable for a ram downloader. This command line option – in combination with -x or -s – allows to create hex files for downloading to ram without the use of the #target directive.

-z --clean

This requests zasm to erase all cached files from the c compiler. These files are stored in subdirectory 's/' in the selected temp directory, which is by default the source file's directory.

-e --compare

This allows to check whether the output for a certain source file has changed: The output file of the assembly is not written but compared to the already existing output file of an older assembly. The main purpose for this option is to support zasm's self test, but you may use it to check your source for code-effective modifications without the need to supply a different output file name and to run cmp against the two files.

```
$> zasm -e zx82.asm "ROMs/zx82 standard rom for ram/zxsp rom for ram.rom"
--> mismatch at $33FC: old=$AA, new=$00
--> mismatch at $33FD: old=$04, new=$00
assemble: 10665 lines
time: 0.1129 sec.
zasm: 2 errors
```

In this example i have compiled the original source of the ZX Spectrum rom and compared it with a modified rom, where a routine which dumps unwanted data at address 0x0000 has been modified to dump it somewhere else, (when i look at the output this seems to be address 0x04AA) so that the rom file can be used in ram (if you have ram at address 0x0000).

If the files match, then simply an ok message is printed and no files are written. If they don't match, then zasm stores a list file with generated opcodes so that you can immediately scrutinize the generated code at the losted addresses.

An output file is actually written to the /tmp/ directory.

-g --cgi

Assemble potentially malicious sources, e.g. in a CGI script for a web site.

This option prevents #insert and #include to include files or directories from outside the source file's directory. Additionally option -I is rejected in #cflags. Traversal via symlinks to any location are possible, also paths in the command line arguments are not affected. #cpath can only be used to select the compiler (without directory path) but only sdcc and vcc are recognized.

--maxerrors

```
--maxerrors=NN
```

Set the maximum amount of errors after which zasm will bail out. Default value is 30 errors. Some errors, especially after assembler directives starting with '#', are 'fatal' and will always immediately terminate assembly regardless of this setting.

--date

```
--date=DATESTRING
```

Set date and optionally time for the __date__ and __time__ macros. This can be used to create reproducible binaries, e.g. for test and verify. The date is also used in the S0 record of Motorola S19 hex files.

Examples:

```
--date=20201231
--date=2020-04-12,12:00:00
--date='2020-4-12 12:00'
```

-o0

Use this if you don't want the output file to be created, e.g. if you just want to recreate the list file. This will save you some milli seconds…

-l0

Use this option if you don't want a list file to be created. Normally, if no list file specific option is given, a 'plain' list file is genertaed which is mostly the same as the source file, plus any included files, plus markers for any occured error.

--z80

Set the target cpu to Z80. This is the default, except if --asm8080 is used to assemble source using 8080 assembler syntax. Alternative: Use the pseudo instruction '. z80' at the start of the source file.

If target Z80 is set with --z80 or .z80 and not by default only, then the label _z80_ is defined and can be tested with 'defined(_z80_)' in expressions. (changed in version 4.3.4)

--z180

Set the target cpu to Z180 / HD64180. This enables additional instructions and disables all illegal instructions, because the Z180 traps them.

Alternative: Use the pseudo instruction '. z180' at the start of the source file.

If target Z180 is set with --z180 or .z180, then the label _z180_ is defined and can be tested with 'defined(_z180_)' in expressions.

--8080

Set the target cpu to the Intel 8080. This disables all additional Z80 instructions. The source must still be in Z80 syntax unless you also use --asm8080, in which case you don't need --8080 because it is implied. Use this option if you want to verify that your program will also run on the 8080 cpu, e.g. for CP/M.

Note: The pseudo instruction '.8080' has an additional effect: '.8080' also selects 8080 assembler syntax!

If target 8080 is set with --8080 or .8080, then the label _8080_ is defined and can be tested with 'defined(_8080_)' in expressions. (changed in version 4.3.4)

--asm8080

Announce that this is an assembler source in the Intel 8080 assembler format with these 'weird mnemonics'. This option implies the option --casefold and set's the default cpu target to the Intel 8080.

'--asm8080' can be used in combination with '--z80' to assemble source for a Z80 (including the Z80's additional opcodes) in 8080 assembler syntax. See section about 8080 assembler instructions. Alternative: Use the pseudo instruction '.asm8080' at the start of the source file.

Note: The pseudo instruction '.8080' has a similar effect but does not only select 8080 assembler syntax but also set's the target cpu to Intel 8080.

--convert8080

Since version 4.3.0: Convert 8080 assembler source file to Z80 style.

zasm converts the source and writes it to the output file or a file with a derived name. Then it assembles the original source and the converted source and compares the outputs. The Z80 source is assembled with option '--casefold'. Flags required to assemble the 8080 source – e.g. '--reqcolon' – must also be used for the conversion. The source is converted independently of any successive errors.

-v[0,1,2]

Set verbosity of messages to stderr. Default is -v1.

By default some warnings, all errors and a '123 errors' / 'no errors' message are printed.

With -v2 some more warnings are printed.

With -v0 nothing is printed.

--ixcbxh, .ixcbxh, _ixcbxh_
--ixcbr2, .ixcbr2, _ixcbr2_

Related command line option, pseudo instruction and predefined label IXCB is used here as a mnenonic for the prefix IX or IY plus prefix 0xCB instructions like 'rr', 'bit', 'set' and 'res'. These are allowed for those opcodes which use the '(hl)' register only. But people have found out what they do when you use one of the other, regular registers: They may additionally copy the result into the selected register or they may, if you use register 'h' or 'l', access the upper or lower byte of the index register.

What a CPU does depends on it's hardware. Original Zilog Z80 CPUs should behave as for --ixcbr2. Others allow the access to the index register halves, some just behave as for '(hl)' for all other registers as well and some, like the Z180, even trap all illegal instruction.

Option --ixcbr2 enables the 'second register target' opcodes like

```
      set 0,(ix+0),b
```

and option --ixcbxh enables the 'index register halves' opcodes like

```
      set 0,xh
```

Obviously they are mutually exclusive.

If command line option --ixcbr2 or pseudo opcode .ixcbr2 was used, then the label _ixcbr2_ is defined and can be tested with 'defined(_ixcbr2_)' in expressions.

If command line option --ixcbxh or pseudo opcode .ixcbxh was used, then the label _ixcbxh_ is defined and can be tested with 'defined(_ixcbxh_)' in expressions.

--dotnames, .dotnames

Related command line option, pseudo instruction and predefined label

Allow label names to start with a dot.

First dots were not allowed in label names. Then some assemblers prepended a dot to the pseudo instructions to easily distinguish them from normal instructions and label names, so they could start at any column of a source line. Having seen that, some people immediately developed the need to start their label names with a dot as well…

Normally, without this option, only pseudo instructions like '.ds' or '.org' can start with a dot. Then they may be written in any column of a source line regardless of the '--reqcolon' setting.

With this setting the limitations described at '--reqcolons' also applies to pseudo instructions starting with a dot because the dot no longer makes them special and implicitely a non-label.

The following example is only valid source with --dotnames and --reqcolon:

```
      .org    0               ; starts in column 0 => requires --reqcolon    .
           ds 0x66            ; one of the original dot named pseudo instructions
           jp .nmi_handler    ; requires --dotname
```

Of course you could clean up your source as well…

Prior to version 4.3.4 a label _dotnames_ was also defined.

`--reqcolon, .reqcolon`

Related command line option, pseudo instruction and predefined label

Select syntax variant where colons ':' are required after program label definitions.

Normally colons are not required and the assembler decides whether a name is a label definition or something else by the position where it starts: label names must start in column 1 and instructions must be preceded by some space.

This example must not be compiled with --reqcolon or --dotnames:

```
.org 0                  ; due to the dot this pseudo instruction is allowed in
                          column 1
foo     equ 0xF00       ; label definition
bar:    and a           ; program label def and a instruction
        jp  foo         ; instruction
shelf   db  0           ; program label def and a pseudo instruction
```

But some source files don't obey this widely accepted rule and put instructions in column 1 or – actually seen – start label definitions indented with some tabs.

As long as only pseudo instructions starting with a dot are put in column 1 and option --dotnames is not required, this is no problem. In all other cases you need this option --reqcolon and, of course, then colons are required after program label definitions, but still not after other label definitions with ' equ' and the like:

This example must be compiled with --reqcolon:

; funny indented:

```
org 0                   ; 'org' would be recognized as a label without --reqcolon
    foo     equ 0xF00   ; label definition
    bar:    and a       ; program label def and a instruction
jp  foo                 ; instruction
    shelf:              ; program label definition
.db 0                   ; a pseudo instruction
```

Prior to version 4.3.4 a label _reqcolon_ was also defined.

`--casefold, .casefold`

Related command line option, pseudo instruction and predefined label

Tell the assembler that label names are case insensitive and that the source does not distinguish between uppercase and lowercase writing of label names.

Note: instruction and register names are always case insensitive.

This option is implied by '--asm8080'.

Normally label names are case sensitive or people just tend to just write them allways the same.

Prior to version 4.3.4 a label _casefold_ was also defined.

`--flatops, .flatops`

Related command line option, pseudo instruction and predefined label

Evaluate expressions strictly from left to right, disobeying any operator precedence.

Old assemblers sometimes didn't know about operator precedence and were designed this way for simplicity.

Use this option if error messages about byte values or offsets out of range are reported by zasm and these expressions make more sense when evaluated this way.

Normal operator precedence in zasm is:

```
unary ! ~ + –  ▶️  >> <<  ▶️  & | ^  ▶️  * / %  ▶️  + –  ▶️  > >= etc.  ▶️  && ||
▶️  ?:
```

Note that this is not the precedence as defined for C. B-). The hierarchy of operators in c is very bad.

Prior to version 4.3.4 a label _flatops_ was also defined.

`-T --test`

Until version 4.1.4: Run zasm's built-in self test. This requires that you have zasm's project directory at hand, because zasm will compile a bunch of source files and compare the output with the files present there. You can append the directory path of the project to the command, else zasm will use the current working directory. You can also append the path to the c compiler with '-c'. If no c compiler is given, then zasm looks for sdcc in subdirectory sdcc/bin/ or sdcc/bin-Linux32/ or sdcc in your $PATH. If the c compiler cannot be found, then some testcases will fail. (currently one testcase.)

```
$> zasm -T ./Test
```

Since version 4.1.5: Run all regression tests in a directory. The directory is scanned recursively. Only files which end with ".asm" and which start with a Shebang '#!' in line 1 are assembled and compared. Symbolic links, whether files or folders, are ignored.

A single regression test case consists of a proper assembler source with a shell-style first line, which was (or could have been used) to assemble this file originally, and the output file created that way.

Formerly the arguments in line 1 are not evaluated by zasm (except if running the regression tests). They are evaluated by the shell, if you run this file as a command. Since version 4.3.4 zasm itself also evaluates some of these options. You may use this trick and always execute your source files to save you some typing in your own projects.

```
$> # create test case from file test1.asm
$> # edit test1.asm and
set line 1 to:
     #!/usr/

local/zasm                        # without any option or
     #!/usr/

local/zasm --
z180 -o original/                 # add options and a output directory
$> cd /my/test/dir                # must stay in source's dir when creating the
reference file
$> chmod u+x test1.asm            # make source executable
$> ./test1.asm                    # execute it to create the reference file
$> # later:
$> zasm -T                        # run all tests in the current directory or
$> zasm -T /my/test/dir           # in the named directory or
$> zasm -T -v                     # with additional options for all test cases
zasm: +++ Regression Test +++
zasm: scanning directory for test sources ...
zasm: found 3 test source files

assemble file: /pub/Develop/Projects/zasm-4.0/Test/8080/zasm-test-opcodes-8080.asm
assembled file: zasm-test-opcodes-8080.asm
    412 lines, 1 pass, 0.0037 sec.
    no errors

assemble file: /pub/Develop/Projects/zasm-4.0/Test/
Z180/zasm-test-opcodes-180.asm
assembled file: zasm-test-opcodes-180.asm
    1999 lines, 2 passes, 0.0121 sec.
    no errors

assemble file: /pub/Develop/Projects/zasm-4.0/Test/

Z80/CPM22.asm
assembled file: CPM22.asm
    3741 lines, 2 passes, 0.0171 sec.
    no errors

total time: 0.0329 sec.
zasm: no errors
$>
```

A regression test file must successfully assemble and produce an output file.

Line 1 is parsed by zasm when running the regression test. Therefore no sophisticated Bash tricks should be used here.

If no output directory is specified in line 1, then the source file's directory is used. The directory path may be a partial path. Then the regression test uses the source file's directory as the starting point. Therefore you should cd into the source directory before running the source to create the reference file, to make sure that the regression test can find it.

There are two unusual instructions which are specially useful in these tests, though they may be used in any source:

Check an assumption:

```
    #ASSERT my_boolean_value
```

Use '!' in column 1 to check that a source line fails:

```
    !   sub de,bc
```

`-c /path/to/cc`

If you include c sources and if the c compiler cannot be found in your $PATH or if you want to use a different one, then you can tell the assembler which executable to use for compiling c sources.

If a partial path is used then your current working directory applies.

`-t /path/to/dir`

Define the directory to use for temporary files. The only temp files created by zasm are those generated by the c compiler which will be stored in subdirectory 's/' inside this filder.

The default location is the output file's directory.

If a partial path is used then your current working directory applies.

`-I /path/to/dir`

Define the path to system header files for the c compiler. Normally the c compiler already knows where they are, but if you want to use a different directory or if your headers just are not in the standard location, you can use this option. It will pass two arguments to the c compiler:

/bin/sdcc --nostdinc -I/path/to/dir

The first one tells it to forget about the standard location for header files and the second one tells it where they actually are.

If a partial path is used then your current working directory applies.

Note: include directories can also be added with the assembler directive

#CFLAGS in your source file. There a partial path refers to the source file's directory!

Note: mind the space!

`-L path/to/dir`

Define the path to the system's library directory. This directory is used by zasm to resolve missing symbols in assembler directive '#include system library'.

The default location is derived from the include directory path. If the include directory path was not specified on the command line then the library path must be specified or ' #include system library' won't work. But you still can hard code the path in the #include statement.

If a partial path is used then your current working directory applies.

Note: mind the space!

Assembler directives

#target

#target <name>

The #target directive defines the kind of output file. If no #target directive is given in the source, then zasm will default to a rom file. The #target directive must occur before any #code definition and any opcode. Best put it directly at the start of the source. The <name> defines the extension of the output file. Depending on the desired target your source must consist of multiple #code segments. See section about target files. The #target directive is required if you include c source files, because the c compiler requires code and data segments.

<name> is one of:

```
ram     plain binary data, presumably to be loaded into ram
bin     old name for target 'ram'
rom     plain binary data, presumably for a rom
sna     ZX Spectrum NMI snapshot
z80     ZX Spectrum snapshot
tap     ZX Spectrum or Jupiter Ace tape file
o       ZX80 tape file
80      ZX80 tape file
p       ZX81 tape file
81      ZX81 tape file
p81     ZX81 tape file
ace     Jupiter Ace snapshot file
```

Only target ram and rom can be written as Intel hex or Motorola S-Record file. All other file types can only be written as binary file. This is controlled by the comand line options '-x' or '-s'.

Simple source file example:

```
#target rom
;
#code   _ROM0,0,4000h
;
; <-- your code goes here -->
;
#end
```

#code

Flag Byte, Fill Byte, Re-enter code segments

```
#code <name>,<start>,<size> [,FLAG=<flagbyte>] [,SPACE=<fillbyte>]
#code <name>,<start>,<size> [,<flags>]    ; old syntax
#code <name>,<start>,<size>
```

This defines a code segment where the generated code will be stored. If the assembler directive #target has been used then at least one #code segment must be defined. Depending on the desired target your source must consist of multiple #code segments. See the section about target files.

Arguments may be left unspecified from right to left. If you need to specify the size but not the address, you can use '*' to mark an argument as unspecified, e.g.:

```
#code   BLOCK1, *, 0x400    ; appended to the previous code block, size = 1kB; no
flag
```

If you define more than one code segment, then the segments (and thus the code therein) will be appended in the output file in the sequence of definition. Typically you define all used segments at the start of your source file in the sequence they shall be appended to the output file and then later re-enter the segments as required. The following is an example as it might be used if you also include c sources:

```
#code   _BOOT,0x0000        ; segment with Reset, RST and NMI vectors
#code   _GSINIT             ; init code: the compiler adds some code here and there
                              as required
#code   _HOME               ; code that must not be put in a bank switched part of
                              memory.
#code   _CODE               ; most code and const data go here
#code   _INITIALIZER        ; initializer for initialized data in ram
```

The name can be chosen freely, but must be a valid label name. zasm generates a label for the start address, end address and length (since 4.0.24) of each segment. If you include c source, then the c compiler requires the above code segments to be defined. Names for segments are case sensitive like all labels, except if command line option --casefold was used.

```
#code FOO, 100, 1000
```

will define these labels:

```
FOO      = $0064 =    100
FOO_end  = $044C =   1100
FOO_size = $03E8 =   1000
```

The start address is required for the first segment only or it will default to 0. Following segments will be automatically assigned start addresses without gap if no address is defined.

The start address should define the 'physical' address for the segment, the address where it is visible to the cpu. It also sets the 'logical' address (the org) for the code which can be 'shifted' with .phase and .dephase.

If a start address is given and does not exactly match the end address of the previous code segment, then the following segments is still appended without gap in the output file. It is assumed that the code will be moved to this address before it is executed.

If the code is written to a .hex or .s19 file, then it depends on the #target, how exactly this 'gap' is handled, because these formats also store the destination address of the contained code.

- For #target rom zasm stores consecutive addresses, suitable for an eprom burner, without gap.
- For #target ram zasm stores the 'physical' address, as set in the #code directive, suitable for a ram loader, which will probably load the code to the 'physical' address of the segments.

You can use the start address if you have a rom which is paged into the Z80 address space, e.g. a 32k rom which consists of 2 pages might be defined like this:

```
#target rom
#code   PAGE1,0,0x4000      ; boot rom
#code   PAGE2,0,0x4000      ; basic rom
```

If given, the size defines the size for this segment. If you store less bytes in it, then the segment will be padded up to this size with the default fillbyte: 0xff for rom and 0x00 for any other target. If your code exceeds the size, then the assembler will generate an error.

Since version 4.4.2 it is also possible to define a custom fillbyte for code segments by appending a key/value pair SPACE=<value> after the size. If you want to leave the size unspecified, use '*'.

If no size is defined, then the segment is exactly as long as the code stored into it.

Flag Byte

The 3rd argument is not required and actually not allowed in most cases. But some targets require a flag for the code segment, e.g. #target tap requires a flag byte for each tape block.

Since version 4.4 it is recommended to define the flag byte using a key/value pair FLAG=<value>.

```
#code HEADER, 0, 17, FLAG = 0x00
```

Fill Byte

Since version 4.4.2 it is possible to define a custom fillbyte for code segments by appending a key/value pair SPACE=<value> after the segment size. This fill byte is used to fill the gap in 'defs' and 'org' pseudo instructions, and for the padding at the end of the segment, if a fixed size was specified.

```
#code _INITIALIZER, *, *, SPACE = 0x00
```

The above example might be used in a rom which incorporates a segment with global data initializers, so that parts of variables, which are set with 'defs' are set to 0x00.

Re-enter code segments

A code segment can be re-entered by using the #code directive with the name only or by using the .area pseudo opcode:

```
#code _HOME         ; following code will be stored in code segment _HOME
      .area _CODE   ; following code will be stored in segment _CODE
                      (sdcc syntax)
```

You can switch between code segments whenever you like in your source. A typical application is initialization code:

You define a segment for init code, e.g. _GSINIT and a segment for most other code, e.g. _CODE. Then at any point in your source where you have some data which need initialization you temporarily switch to the _GSINIT segment and add the init code. This is what the c compiler does for initialized variables.

#data

```
#data <name>,<start>,<size>
```

This directive declares an area of ram to be used for variables etc. #data behaves much like #code, except that the contents of this segment is not stored into the output file. The segment is used merely to define labels, presumably in ram.

For a simple rom written entirely in assembler you probably need only one data segment, or even no data segment at all, if you just define the variables' addresses with the pseudo instruction equ.

The name can be chosen freely, but must be a valid label name. zasm generates a label for the start address, the end address and the size of each segment (since 4.0.24). Names for segments are case sensitive like all labels, except if command line option --casefold was used.

```
#data DATA, 100, 1000
```

will define these labels:

```
DATA      = $0064 =    100
DATA_end  = $044C =   1100
DATA_size = $03E8 =   1000
```

Typically the assembler instruction 'defs' alias ds' is used to allocate space in a data segment. You cannot use instructions which actually try to store bytes into this segment – only gaps.

In addition the assembler instruction 'data' is allowed in a data segment, which just does the same.

Also allowed are org and align, because they also just insert space, they're just different in how they calculate the gap size.

The following example might be used if you include c source:

It defines two data segments named _DATA and _INITIALIZED, which is where the c compiler stores uninitialized and initialized variables. In this example these segments are fittet into the 'printer buffer' of the ZX Spectrum, a mostly unused area (except for printing) of 256 bytes.

The segment _INITIALIZED is defined with address '*' which means that it's start address will be directly appended to the previous segment's end, and the equation for the size makes sure that the combined size of both segments is exactly 256 bytes and allocating more variables will be detected by the assembler and raise an error.

```
#data   _DATA, printer_buffer                       ; uninitialized data
#data   _INITIALIZED, *, 256 - (_INITIALIZED-_DATA) ; data initialized from
                                                      _INITIALIZER
#data   _HEAP, code_end     ; heap:
__sdcc_heap_start:          ; --> sdcc _malloc.c
        ds  ram_end-$-1    ; add all unused memory to the heap
__sdcc_heap_end:            ; --> sdcc _malloc.c
        ds  1
```

A data segment can be re-entered by using the #data directive with the name only or by using the .area pseudo opcode:

```
#data _DATA         ; following code will be stored in data segment _DATA
      .area _DATA   ; following code will be stored in segment _DATA   (sdcc
syntax)
```

You can switch between segments whenever you need to allocate storage space for variables in your source.

```
...
#data   _DATA       ; switch context to data segment _DATA
foo     defs    4
bar     defs    2
fuzzy   defs    $20
;
#code   _CODE       ; switch context back to code segment _CODE
...
```

The c compiler uses the code segment _INITIALIZER to store the initial data of initialized variables. These are copied during system initialization to the location of _INITIALIZED. (Actually the c compiler doesn't do this. It is your job to add some code in the _GSINIT segment to copy the initializer data!) So you can switch between these segments to allocate the variable in ram and add init data in rom:

```
...
#data   _INITIALIZED
foo     defs    2       ; allocate space in ram
#code   _INITIALIZER
        defw    4711    ; store init data in rom. SIZE AND POSITION MUST MATCH!
#code   _CODE
…
```

#test

#test <name>,<start>,<size>

Define code segment for automated tests after successful assembly.

The test code will be loaded on top of the assembled code and executed. During execution i/o addresses are monitored and compared with supplied values. Finally – or at any time during the test – registers and the cpu cycle count can be compared against expected values.

There can be any number of tests in a test segment and there can be any number of test segments which are loaded and executed separately. Test errors are handled like assembly errors. Only if all tests pass the assembled code is finally written.

Technical Specification of the Test Code Runner

CPU emulation

zasm can emulate the 8080, Z80 and Z180 cpu.

The instruction set is limited to the selected CPU, eventually modified by command line options --ixcbr2 or –ixcbxh. If an unimplemented / illegal instruction is executed, the emulation stops with an error and the test fails.

8080 emulation

The 8080 cpu has no unhandled/undocumented opcodes, but some opcodes have aliases which were used by the Z80 for additional functions, e.g. relative jumps, 2nd register set and prefix opcodes.

Only the official opcodes, which are also those which are generated by zasm, are allowed. Execution of an alias opcode, which was reused by the Z80, is handled like an illegal opcode and will terminate the test. This is to prevent you from accidentially writing code which will not run on a Z80.

Z80 emulation

- Widely used undocumented opcodes are always executed without generating an error. These are SLL and access to the upper and lower half of index registers in LD and arithmetical/logical operations.
- Illegal index register opcodes after after prefix IX/IY + prefix 0xCB are allowed, if the corresponding command line option --ixcbr2 or --ixcbxh was set. Then they are performed according to this setting. [TODO: the timing of ixcbih opcodes is unclear]

- All other undocumented / illegal opcodes are rejected and will terminate the test with an error.

Z180 emulation

All Z180 opcodes are emulated and the different timing is implemented. The Z180 traps illegal instructions and so does the emulation. [TODO: the exact behavior of the SLP opcode is unclear]

The test code is executed by a variant of Kio's Z80 engine.

The emulation speed is approx. 800MHz@Z80 per GHz@Host (2.6GHz on 3.2GHz AMD Ryzen5 2400G) or let's say, depending on the emulated system and the speed of your host, the emulated Z80 is running 1000 times faster than the original. :-)

I found out that the emulation speed is very sensitive to the loading position of the Z80 engine and can vary widely - for my PC from 1MHz emulated speed to 2.5MHz! I'll try to check the speed before releases of zasm but will probably frequenty fail to do so. After all it can be completely different on someone else's PC.

The emulation of the Z80 is very precise but i cannot guarantee correctness to the last bit and cycle. If you find any difference please report a bug. Replication of the undefined behavior of flags is 80% exact.

Memory, interrupts and i/o

The assembled code is loaded into 64kB ram, each segment at the address defined in the #code directive, ignoring possible overlap. Finally the test segment is loaded, again happily overwriting already loaded segments.

- Memory mapping is not supported.
- Wait states are not supported.
- Memory mapped i/o is not supported.

The emulation can be run at maximum speed or limited to a fixed frequency.

A timer interrupt can be enabled and used in multiple ways:

- A timer interrupt after a fixed number of cpu cycles (cc)
- A timer interrupt with a given frequency when the cpu is running at a fixed speed
- A timer interrupt with a given frequency when the cpu is running at maximum speed
- Other interrupt sources are not supported.
- The non-maskable interrupt NMI is not supported.

A timeout for a test can be specified in realtime milliseconds.

In and Out access is monitored and must be prepared for. Access to an unprepared address immediately terminates the test.

I/o addresses can be prepared in multiple ways:

- Assign an address to the console (stdin+stdout)
- provide input data for an In address
- provide test data for an Out address

Not yet supported in zasm v4.4.4

- Loading of a rom file
- read/write/compare to file

Setting up a test case

```
#test <name>, <addr>, <size>
;
.test-clock     <frequency>
.test-int       <frequency>  [, <duration>]
.test-int       <cpu_cycles> [, <duration>]
.test-intack    <int_ack_byte>
.test-timeout   <duration>
;
.test-in        <addr>, <values>
.test-out       <addr>, <expected_values>
.test-console   <addr>
```

.test-clock

Define a fixed speed for the emulation. The emulation will be throttled to the given frequency.

If .test-clock is not specified or if the clock is set to 0 then the test runs at the fastest possible speed.

```
.test-clock 4000000 Hz ; run at 4 MHz.   mind the space!
.test-clock 3500 kHz   ; run at 3.5 MHz. mind the space!
.test-clock 6 MHz      ; run at 6 MHz.   mind the space!
```

.test-int

Enable a timer interrupt and define the interrupt frequency.

The frequency can be given in cpu cycles 'cc' or in real-world frequency. Depending on whether the cpu is running at a fixed speed or unlimited speed the executed cpu cycles per interrupt are predictable or may vary widely.

```
.test-int  50 Hz       ; interrupt with real-world frequency
.test-int  70000 cc    ; interrupt after 70000 cc (50 Hz at 3.5 MHz: the ZX
Spectrum :-)
.test-int  50          ; interrupt with 50 Hz
.test-int  70000       ; interrupt after 70000 cc
```

The units cc or Hz are optional. If present, a sanity test is included.

If the unit is omitted, zasm uses values up to 1000 as Hz and larger values as cc.

1000 is the highest value which can be used for frequency.

Since version 4.0.3 a second argument can be added to specify the duration, for which the interrupt should be asserted. This can be used to test whether interrupts are missed due to longisch sequences with interrupts disabled or whether they interrupt itself, if enabled in the interrupt handler too early.

It is recommended to use at least 32 cc for the duration, because evenif your code never disables interrupts (which will easily take longer than that) the longest Z80 opcodes takes 23 cpu cycles! And, of course, best use the actual duration of your real system. :-)

If no interrupt duration is specified, then the interrupt is asserted until it is acknowledged (handled).

```
.test-int  50 Hz, 40 cc    ; interrupt with real-world frequency, asserted for 40
                             cc
.test-int  70000 cc, 48 cc ; interrupt after 70000 cc, asserted for 48 cc
.test-int  50, 128         ; interrupt with 50 Hz, asserted for 128 cc
.test-int  70000, 64 cc    ; interrupt after 70000 cc, asserted for 64 cc
```

There is one more subtable difference between auto-off mode and int-with-duration mode:

In auto-off mode the first interrupt, which would occur at cpu cycle cc = 0 is suppressed.

In interrupt-with-duration mode the first interrupt at cc = 0 is not suppressed, and if the test code enables interrupts very early, then the first interrupt is immediately taken.

.test-intack

Define byte to be read from the bus in an interrupt acknowldge cycle.

This byte is 0xFF by default (floating bus on a TTL system) but different values can be defined as provided by some peripheral hardware, to supply a RST opcode or an index in the interrupt vector table, depending on the current interrupt mode of the cpu.

```
.test-intack  255-8           ; this would be a RST 48
.test-intack  opcode(rst 48)  ; the same, just readable :-)
.test-intack  lo(int_vector)  ; low byte of memory address of an interrupt vector
                              ; the high byte would go into the i register.
```

.test-timeout

Define a timeout for the test. If the test takes longer it is aborted and fails.

The timeout is measured in realtime.

If no timeout is set or the timeout is set to 0 then there is no timeout.

```
.test-timeout  1 s     ; timeout after 1 second.        mind the space!
.test-timeout  500 ms  ; timeout after 500 millisecond. mind the space!
.test-timeout  1 m     ; timeout after 1 minute.        mind the space!
```

.test-console

Attach an i/o address to stdin and stdout.

This is an easy way to print messages from the running code.

A little more effort must be taken to input text, because the input is not blocking.

If no input is available, then the port reads as 0x00.

```
.test-console  0xFE    ; use port address 0xFE,   high byte is ignored
.test-console  0xFFFE  ; use port address 0xFFFE, you must use in(c) and out(c)
```

.test-in

Supply input data for an input port. The supplied data is read by inputs from this port.

The test fails, if the test code reads more data than supplied and it fails if there is some unread data left after the test.

Basic example

```
.test-in  0xF8, 0,0,0,"Hello world!",13,0,0,0
```

This prepares port 0xF8 to return 3 times 0x00 (probably meaning 'no data'), then a text and finally again 3 times 0x00. All data must be read by the test code.

```
.test-in  0xF8, 0,0,0,"Hello world!",13,0,0,0
.test-in  0xF8, 0,0,0,"Here i go again!",13,0,0,0
```

This prepares port 0xF8 with more data. The input data is combined from multiple definitions.

Repeated data

You can repeat blocks of data:

```
.test-in  0xF8, {0}*10, "Hello World!", 13, {0}*10

.test-in  0xF8, {0}*10, {"Hello", 32, "World!", 13} * 5, {0}*10
```

You can add a final block to repeat forever. This won't make the test fail:

```
.test-in  0xF8, {0}*10, "Hello World!", 13, {0}*
```

.test-out

Supply output test data for an output port. The supplied data is compared against data written to this port.

The test fails, if a byte written doesn't match the expectation, if the test code writes more data than supplied and it fails if there is some unwritten data left after the test.

Basic example

```
.test-out  0xF8, "Hello world!",10
```

This prepares port 0xF8 to expect a text and a linebreak. All data must be written by the test code.

```
.test-out  0xF8, "Hello world!",10
.test-out  0xF8, "Here i go again!",10
```

This prepares port 0xF8 with more data. The output test data is combined from multiple definitions.

Repeated data

You can repeat blocks of data:

```
.test-out  0xF8, {'-'}*12, 10, "Hello World!", 10, {'-'}*12, 10

.test-out  0xF8, {'-'}*12, 10, {"Hello World!", 10} * 5, {'-'}*12, 10
```

You can add a final block to repeat forever. This won't make the test fail:

```
.test-out  0xF8, {"Hello World!",10,}*  ; any number of repetitions of this block
```

```
.test-out  0xF8, "Hello World!", 10, *  ; any data whatever after text
```

Run test code and test your expectations

After the test segment is setup, you can add your test code. It starts up with all registers set to 0x00 and interrupts disabled. Typically you load some registers and call a function to test. It is recommended to wrap all tests in a #local context so you can reuse label names:

```
#local
    ld  sp,0
    ld  a,3
    ld  de,7
    call A_times_DE
    ;
.
expect a' = 0
.
expect a = 3
.
expect de = 7
.
expect hl = 3*7
.
expect dehl = 7*$10000 + 3*7
.expect cc < 1000


#endlocal
```

.expect register

.expect can be used to test register values.

You can compare the value of any register, register pair or quad register to match an expected value. Ordered comparisons ('>' or '<') are not allowed.

Recognized register names are:

```
 a f b c d e h l a2 f2 b2 c2 d2 e2 h2 l2 a' f' b' c' d' e' h' l'
xh xl yh yl ixh ixl iyh iyl pch pcl sph spl i r
af bc de hl af2 bc2 de2 hl2 af' bc' de' hl' ix iy pc sp
bcde bchl bcix bciy debc dehl deix deiy etc.
im iff1 iff2
```

.expect cc

.expect can also test the cpu cycle counter.

You can compare the value of the cpu cycle counter to match an expected value or use an ordered comparisons '>' … '<'.

The cycle counter is reset after each block of .expects, either cc or registers.

```
.expect cc = 456
.expect cc >= 400
.expect cc > 400
.expect cc <= 500
.expect cc < 500
```

Tips

Hint: assemble your source with command line option -uwy to see the generated code and cycle count.

Hint: run tests with command line option -v to see more interesting output. B-)

#if, #elif, #else, #endif

```
#if <condition>
    ...
#elif <condition>
    ...
#else
    ...
#endif
```

#if starts a block of assembler instructions, which is only assembled

if the given <condition> is true. The <condition> must be evaluatable in pass 1. Conditional assembly may be nested. Any assembler directive except pairing #else, #elif and #endif, and any assembler instruction or pseudo instruction is skipped if the <condition> is false. If the pairing #endif is not found until the end of file then a nesting error is assumed. Note, that #include is also skipped if <condition> is false.

As an exception the pseudo instructions 'if' and 'endif' are detected and handled just like '#if' and '#endif'. note: this may change.

After an #if directive where the <condition> was false, the #elif <condition> is checked and if it is true, the following block is assembled. If it is false, or if already a block was assembled within this #if ... #endif range, the following block is skipped. #elif is optional and may occur multiple times after #if.

After an #if directive, #else negates the <condition>. If it was false it becomes true and the following assembler instructions are assembled. If it was true it becomes false and the following assembler instructions are skipped. #else is optional and should only occur once before the #endif assembler directive. If #else occurs after some #elif directives, then the #else part is a final 'catch all'.

The #endif directive finishes conditional assembly as started with the #if directive.

#end

```
#end
```

Define the logical end of your source.

#end must not be inside an #if branch or in a #local scope. This is so that using #end does not disable the test for a missing #endif or #endlocal.

Anything after #end is ignored by the assembler.

Alternate syntax:

zasm also accepts 'end' as a pseudo instruction:

```
	end
```

or

```
	.end
```

#include

Include a single source file or resolve missing labels from a library.

#include <filename>

```
	#include "utils.ass"       ; file in same directory
	#include "./utils.s"       ; file in same directory
	#include "utils/u1.s"      ; file in subdirectory "utils/"
	#include "main13.c"        ; include a c source file
```

Include a single source file. The file name must be an absolute path or relative to the current source file. The included source is assembled as if it was in the main source file itself. Source file inclusion may be nested.

The included file may be a c source file, which is detected by the filename extension ".c". Then zasm executes sdcc (or any other c compiler) to compile the source.

Topics related to c files:

- Command line option -c may be used to point zasm to a certain compiler.
  If not used, zasm will search in your $PATH for an executable named 'sdcc'.
- Command line option -I may be used to supply the path to the system headers library.
  If not used, sdcc will use it's built-in default paths.
- Command line option -L may be used to supply the path to the system library sources.
  This is used by zasm if you use #include standard library. (see below)
- Directive #CFLAGS may be used in your assembler source to tweak the cflags for the c compiler.
- zasm surrounds included c files with the #local and #endlocal directives to avoid problems with identically named static variables.

#include library

Resolve missing labels from a library.

```
	#include library <"directory">
	#include library <"directory"> resolve *
	#include standard library
	#include standard library resolve _foo, __bar, __mulu16
```

#include library or #include standard library may be used to automatically resolve undefined labels. #include standard library requires that you have passed the system library directory with command line option -L.

You can either try to resolve all missing labels, in which case you don't append the resolve keyword or resolve *, or you may limit automatic label resolving to a list of certain labels. This may be required if you have to spread the additional code over multiple code segments or if certain labels must go into certain code segments and the included sources do not care for this by themselves.

This is most conveniant to automatically resolve references arising from c source, but may be used with pure assembler source as well.

Naming convention for files in a library directory

Files in the library directory must follow certain naming conventions: they must be named according to the label they define. If a file defines label _foo, then the file must be named _foo.s or _foo.c, or more accurately any _foo.* will be included. If it ends with '.c' then it will be compiled and then included. If a file defines more than one label then you can make symbolic links for the other names which point to this file.

zasm adds directive #assert after the included file to verify, that the promissed label was actually defined:

Source:

```
	#include standard library
```

Generated:

```
	5E30:			#include standard library
	5E30:			#include "/Projects/sdcc/lib/__mulint.s"
	5E30:			; contents from file
	...			...
	7503:			#assert defined(__mulint::)
	7503:			#include standard library
	7503:			#include "/Projects/sdcc/lib/_check_struct_tm.c"
	7503:			#local
	7503:			; contents from compiled file
	...			...
	8D96:			#endlocal
	8D96:			#assert defined(_check_struct_tm::)
	8D96:			#include standard library
	...			...
```

You can see how the single #include standard library is expanded into a series of include filename and an #assert defined(labelname::) is appended to the included source. C sources are also wrapped with #local ...#endlocal.

#include standard library is generated again and again until no more labels could be resolved. (This is the way how it works internally.)

As an alternate syntax the pseudo instructions "include" and "*include" are recognized.

```
		include "foo.s"
		*include "bar.s"
```

#insert

```
	#insert <filename>
```

Inserts a binary file. The file name must be given as an absolute path or relatively to the current source file.

The file's contents are just copied into the code segment verbatim.

Examples:

```
#insert "image.gif"       ; file in same directory
#insert "./image.gif"     ; file in same directory
#insert "pics/g1.gif"     ; file in subdirectory "pics/"
```

As an alternate syntax the pseudo instruction "incbin" is also supported:

```
    incbin "characters.bin"
```

#local, #endlocal, .local and .endlocal

This defines a local scope for normal label definitions. This way you can safely use 'standard' names for jump targets in included files without worrying about name collissions. You can still push labels to the global scope though.

#local contexts can be nested.

Example: Wrap an included source file to avoid label name conflicts.

Note: zasm does this automatically for c files, but not for assembler files:

```
#local
#include "util/u.asm"
#endlocal
```

Define global variables from inside a #local context:

```
#local
boo::   ld a,0      ; '::' makes 'boo' global
        .globl fax
fax:    ret         ; 'fax' is now also global
#define bar 4711    ;
labels defined with #define are also global.
#endlocal
```

Note: used but locally undefined labels are automatically pushed to the surrounding scope by #endlocal and will finally reach the global scope, if they were not defined in an intermediate surrounding #local scope. There they can be picked up and resolved by the #include library directive.

The .globl pseudo instruction can also be used to make sure that a certain label is global and not accidentially defined in a surrounding #local scope.

In some cases you need labels to be resolved in pass 1, e.g. for #IF <cond> or for . rept. Then you run into problems when you refer to global labels from within a #local scope: zasm does not know for shure, whether it should use the valid global label, because a local label with this name still might be defined after this point of use. Therefore all references to global labels from within a local scope are never valid in pass 1. To solve this problem you can declare this label global using the .global pseudo instruction.

#assert

```
#assert <condition>
    .assert <condition>
```

Assert that the given condition is true.

This directive might be used to detect logical or overflow errors in your source. It is used by zasm itself to assert that loaded library files actually defined the label they promissed to define. It is also used in zasm's assembler test suite.

Example:

```
#assert ( $ & 7 ) == 0      ; assert that current code position
                            ; is a multiple of 8

.area GSINIT
.
assert DATA_size > 1       ; else ldir will run amok

ld  bc,DATA_size - 1
ld  de,DATA + 1
ld  hl,DATA
ld  (hl),0
ldir
```

#define

```
#define <name> <value>
```

Define an alias for a well-known pseudo instruction or define a global label.

#define is used in some sources in three ways:

1. Define an alias for an instruction name
2. Define a label
3. Define a preprocessor function with arguments

From that the most common aliases and the definition of labels are supported.

Define an alias

```
#define equ  .equ
#define defs .ds
```

Within the following groups names can be mapped in any direction:

- org .org
- defw dw .dw .word
- defb

- db .
- db .byte
- defm dm .dm .text .ascii
- defs ds .ds .block

All these mappings are recognized and … ignored.

Other mappings result in an error. (if you feel a mapping should be added then send an email.)

zxsp does not support aliases for instruction names, but it already knows about quite a lot of them. Therefore it can ignore these #defines.

Use of this directive is discouraged. It is provided because some sources use it. Just delete it.

Define a global label

```
#define foo 123
#define bar $ - shelf
```

#define can be used to define a global label as with

```
foo:: equ 123
bar:: equ $ - shelf
```

Labels defined with '#define' are always global, even if defined inside a local context.

Use of this directive is discouraged, many assemblers do not understand it. Use the pseudo instruction ' equ' instead.

#charset

```
#charset <name>
#charset none
#charset map <characterstring> = <firstcodepoint>
#charset unmap <characterstring>
```

zasm can translate characters from the source file (which must be ascii or utf-8) to the character set of the target system. If no translation was set, then Latin-1 is assumed for your target system, which simply means: nothing is translated, the source is assumed to be utf-8 encoded and after utf-8 decoding the resulting characters must be in range 0x00 to 0xFF.

Translation should only be applied to printable characters, though it might be tempting to convert '\n' (10) to '\r' (13) for targets which use '\r' to start a new line.

zasm translates character literals like 'A' and strings like "foobar" into the selected character set. If a translation is not available in the target character set, then zasm issues an error.

This translation is done everywhere: in pseudo opcodes like defm as well as in immediate arguments of opcodes or in the value for #if, #code or #data. It is not applied to filenames in #include etc. or to the string in #charset itself.

Caveat: It is applied to the numeric value <firstcodepoint> in #charset, if you use a character literal here.

```
        defm  "Say hello" + 0x80
        ld    a,'D'+1
        defb  0x20, 'a', '▞', '£'
    #assert ' ' == 0                ; is the ZX80 / ZX81 character set selected?
```

This translation is also applied to strings in c sources: sdcc creates .ascii pseudo instructions which are handled just like defm.

But unluckily sdcc emits only hexcode constants for character literals, and for this reason character literals in the c source will not be translated:

```
    static const char foo[] = "My little secret";   // will be translated
    char c = 'A';                                    // won't be translated :-(
```

Using character set translation is especially useful for targets with a character sets which is not based on ASCII, e.g. for the ZX81.

Available translations are:

- zx80
- zx80_inverted
- zx81
- zx81_inverted
- zxspectrum
- jupiterace
- jupiterace_inverted
- ascii
- none

ZX80 and ZX81 have completely non-ascii character codes, even character code 0x00 is a legal character: the space. (caveat: this may lead to problems in c source.) There are no lowercase letters. Instead lower case letters are used as an easy way to create an inverted (uppercase) letter.

ZX Spectrum and Jupiter Ace have a character set which is mostly ascii, but with '©' and '£' added.

They all have block graphics symbols, e.g. these characters can be used in a source file for the ZX Spectrum: "©£ ▝▘▀▗▐▚▜▖▞▌▛▄▟▙█". (note: the ' ' in the string is the non-breaking space.)

ASCII limits available characters to the 7-bit ascii range.

NONE resets the character translation. (effectively to Latin-1, see above.)

The inverted character sets produce characters with bit 7 toggled (bit 7 indicates inversion in all character sets) compared to the not-inversed character set. As the block graphics already consist of inverted and not inverted characters, the inverted characters are reverted to normal in the inverted character sets.

It is possible to enable, redefine and disable the character set throughout the source whenever and as often as required.

Custom character sets

You can construct your own translation table starting from any predefined translation or from scratch:

```
#charset ascii
#charset map "abcdefghijklmnopqrstuvwxyz" = 'A'
#charset unmap "`{|}~"
```

In this example we limit the range of available characters to uppercase characters, e.g. for a veerry oolld printer:

We start with the ASCII character set and map all lowercase characters to their uppercase counterparts and remove all special characters in the lowercase characters vicinity. The resulting characters are now limited to the range 0x20 to 0x5F. Characters 0x60 to 0x7E are either remapped or disabled. Character 0x7F was no ascii character from the start.

Now all uppercase and lowercase characters in character literals and strings in your source are translated to uppercase ASCII characters and any use of '`' to '~' will result in an error.

#compress

```
#compress <segment_name>
#compress <segment_name1> to <segment_name2>
```

new in version 4.1.0

This defines that the named code segment or range of code segments shall be compressed using Einar Saukas' "optimal" LZ77 packer. The named segments must be defined before the #compress directive and they must be #code segments. Compression of #data segments makes no sense, as they are not stored in the output file. zasm currently only supports bottom-up compression, where the decompression increments like in 'ldir'. If multiple segments are compressed then all but the first must not have an explicit address set. They must follow each other without any address gap.

Z80 decompressors are included in the zasm distribution or can be downloaded from Einar Saukas' Dropbox.

Defined Labels

#compress will define some labels for you to use in your source:

#compress SEGMENT

```
SEGMENT_csize
SEGMENT_cgain
SEGMENT_cdelta
```

#compress SEGMENT1 to SEGMENT2

```
SEGMENT1_to_SEGMENT2_size
SEGMENT1_to_SEGMENT2_csize
SEGMENT1_to_SEGMENT2_cgain
SEGMENT1_to_SEGMENT2_cdelta
```

Note: zasm always defines the following labels for all segments:

```
SEGMENT
SEGMENT_size
SEGMENT_end
```

SEGMENT_size and SEGMENT1_to_SEGMENT2_size

This is the uncompressed size of the segment(s).

SEGMENT_csize and SEGMENT1_to_SEGMENT2_csize

This is the compressed size of the segment(s) as stored in the output file.

SEGMENT_cgain and SEGMENT1_to_SEGMENT2_cgain

This is the difference between uncompressed and compressed size. If it's negative then compression actually increased the size of your data.

SEGMENT_cdelta and SEGMENT1_to_SEGMENT2_cdelta

A common szenario is that the compressed data and decompressed code segment overlap and the decompressed data overwrites the compressed data while it grows. Then the compressed data must be loaded high enough above the decompressed data so that no unprocessed bytes are overwritten. Label 'cdelta' defines the minimum difference between compressed data end and uncompressed data end.

Implementation

Compression unavoidably leads to relocated code. So let's start with code relocation first.

Imagine you load code at a 'load_address' and move it down to 'code_address' before it is executed:

```
#code LOADER, load_address
loader: ld      hl, LOADER_end      ; address of
#code MCODE after loading
        ld      de, code_address    ; to be moved here
        ld      bc, MCODE_size
        ldir
        jp      reset               ; = code_address


#code MCODE, code_address
reset:  di
        ld      sp,0
        ...
```

Note: you must be careful to get your label values right. This can be a real headache.

The whole code is loaded to (or initially located at) 'load_address'. Therefore we declare our LOADER segment to start with this physical address, so that all labels inside this segment are right. (currently there's only one label 'loader').

The CODE segment is copied to address 'code_address' and executed there. Therefore we declare our CODE segment to start with this physical address, so that all labels inside this segment are right.

It becomes obvious that source and destination block must not overlap and the loader code must not be overwritten as well, at least not the last 5 bytes: 'ldir' and 'jp reset'.

Since large code loaded from tape can frequently not be loaded high or low enough so that the relocated block does not overwrite the loader, we have to make one first improvement: If we move the block downwards then put the loader at the end else use lddr instead of ldir. The lddr version is not discussed here as it is not yet supported by zxsp. So let's put the loader at the end:

```
#code MCODE, code_address
reset:  di
        ld      sp,0
        ...


#code LOADER, load_address + MCODE_size
loader: ld      hl, load_address    ;
#code MCODE after loading
        ld      de, code_address    ; to be moved here
        ld      bc, MCODE_size
        ldir
        jp      reset               ; = code_address
```

Now let's add compression: This is very similar except that the code on the tape is compressed and therefore the block will grow during 'ldir'.

```
#code MCODE, code_address
reset:      di
    ld      sp,0
    ...


#code LOADER, load_address + MCODE_csize
loader: ld      hl, load_address        ;
#code MCODE after loading
        ld      de, code_address        ; to be moved here
        ;ld     bc, MCODE_size          ; not reqired
        call    decompress_zx7_standard ; ldir
        jp      reset                   ; = code_address

#include "decompress_zx7_standard.s"    ; include the zx7 decompressor

#compress MCODE                         ; define MCODE to be compressed

#assert code_address+MCODE_csize+MCODE_cdelta <= load_address+MCODE_csize

#assert MCODE_end+MCODE_cdelta <= LOADER
```

Mind the difference: the LOADER is no longer located at load_address+MCODE_size but now at load_address+MCODE_csize. Right after loading the bytes from load_address to load_address+MCODE_csize are compressed and not executable at all.

#compress MCODE defines that the code segment 'MCODE' shall be written compressed to the output file.

Finally there's a check to assert that the decompressed code will not overwrite the not yet processed compressed data. Both #asserts are identical in this example.

Resolving Cyclic Dependencies

It's not obvious but zasm resolves a cyclic dependency here: As the LOADER address depends on the compressed MCODE size, all code labels in the LOADER segment are not valid until the compressed MCODE size is valid. But the compressed MCODE size cannot be valid unless all labels in the source are valid, because a not-yet valid label may be used in the MCODE segment and the compressed size may change when the label value is finally resolved. Therefore MCODE_csize never becomes valid.

zasm uses a 'preliminary' state here to solve this problem and therefore requires at least one more pass to assemble the source.

Sometimes 'preliminary' labels may toggle between each pass or infinitely grow and grow and grow, depending on your source. Then assembly fails. The way to solve this situation depends on the kind and location of the problem. Frequently the problem results from code alignments in your source. E.g. there is a '.align 2' and in one pass it adds no space, zasm adjusts the segment address by one and now it inserts a byte, annihilating zasm's effort. Similarly the compressed data size can toggle between two values when a label used in the uncompressed code depends on the compressed data size.

This can eventually be solved by adding a dummy byte. The effect will not last for ever, if it happens again remove the byte.

__date__ and __time__

Normally the assembly success should be predictable. If the uncompressed code contains assembly-time dependent data, as __date__ and __time__, then assembly may succeed or fail unpredictably: you assemble: it fails, you assemble again: it succeeds, you assemble again: it fails, and so on. Using the 1-byte trick may eliminate the problem for a while, but may also just put this data in a uncompressed section instead.

Compression Time

Though the zx7 compression is pretty fast on small code blocks, it can become very slow on larger ones because it has a quadratic time stamp. Additionally larger sources with lots of segments tend to require more passes for the assembler. If assembly time becomes too long then you can do two things:

- Split the compressed blocks. This will speed up assembly time but grow your source, both for additional calls to the decompressor and a not so good compression result.

- Store and compress data in separate files and #insert them. This is the traditional method anyway. This is good for static data with no references to other data but does not work if it contains references to non-const labels in the rest of the program.

Supported Target Files

Compression can be used with most file formats. There are certain restrictions on what can be compressed, e.g. the file headers of snapshot files cannot be compressed. While with TAP files the major benefit is the reduced loading time, the major benefit for snapshot files probably is that you can store more game data like levels.

Currently (vs. 4.1.0) Z80 files cannot be compressed. This will be implemented as soon as possible.

#cpath

```
#cpath "/path/to/c-compiler"
```

Example:

```
#cpath "/usr/bin/sdcc"
#cpath "sdcc"
```

Set the path to the c-compiler from the assembler source file. Normally you set the path from the command line with option -c. The path given on the command line overrides the #cpath declaration in the file. If more than one #cpath directive is given, then only the first is evaluated and all others are silently ignored.

The #cpath directive can also be used to declare the compiler to use (without directory path), either sdcc or vcc. Then the executable will be searched in your search path.

In CGI-Mode this directive is not allowed.

#cflags

```
#cflags <option1> <option2> ...
```

Example:

```
#cflags $CFLAGS --reserve-regs-iy
```

Define or modify the flags passed to the c compiler.

This option is only useful if you include c sources.

#cflags defines new flags for the c compiler. If you just want to add another option, you can include the current flags with the keyword $CFLAGS.

Some flags are always added and cannot be removed with #cflags:

The flags '-S' and '-mz80' for sdcc

An include directory passed with "-I" on the command line.

After your custom flags zasm adds '-o', the destination file name and the source file name. In case you use a different c compiler (i don't know whether this is possible) then you can modify passing of the source and destination file with the keywords $SOURCE and $DEST.

useful flags for sdcc:

```
-Dname[=value]          Define macro
-Ipath                  Add to the header include path
--nostdinc              Do not include the standard directory in the
                        #include search path
--Werror                Treat warnings as errors
--std-c89               Use C89 standard (slightly incomplete)
--std-sdcc89            Use C89 standard with SDCC extensions (default)
--std-c99               Use C99 standard (incomplete)
--std-sdcc99            Use C99 standard with SDCC extensions
--funsigned-char        Make "char" unsigned by default
--disable-warning nnnn  Disable specific warning

--callee-saves func[,func,...]  Cause the called function to save registers
                        instead of the caller
--all-callee-saves      all functions save and restore all registers
--callee-saves-bc       all functions save/restore BC on entry/exit (z80 only)
--fomit-frame-pointer   Leave out the frame pointer.
--fno-omit-frame-pointer  Never omit frame pointer (z80 only)
--stack-probe           insert call to function __stack_probe at each
                        function prologue
--no-c-code-in-asm      don't include c-code as comments in the asm file
--no-peep-comments      don't include peephole optimizer comments
--codeseg name          put code (and const's) in .area _name,
                        e.g. --codeseg CODE => .area _CODE
--reserve-regs-iy       Do not use IY (incompatible with --fomit-frame-pointer)
                        (z80 only)

--nogcse              No GCSE optimisation
--nolabelopt          No label optimisation
--noinvariant         No optimisation of invariants
--noinduction         No loop variable induction
--nojtbound           No boundary check for jump tables (dangerous)
--noloopreverse       No loop reverse optimisation
--no-peep             No peephole assembly file optimisation
--no-reg-params       Don't some parameters in registers (On some ports)
                      TODO: z80?
--peep-asm            Do peephole optimization on inline assembly
--peep-return         Do peephole optimization for return instructions
--no-peep-return      No peephole optimization for return instructions
--peep-file file      use this extra peephole file
--max-allocs-per-node Maximum number of register assignments considered at each
                      node of the tree decomposition
--nolospre            Disable lospre. lospre is an advanced redundancy elimination
                      technique, essentially an improved variant of global
                      subexpression elimination.
--lospre-unsafe-read  Allow unsafe reads in lospre. code may sometimes read from
                      random memory locations. (should be ok for the z80, because
                      it has no memory mapped i/o)
```

#! (Shebang)

```
#!/usr/local/bin zasm --flatops -uwy -x
```

Normally such a line would be the first line in your source file if you make the file executable. Then line 1 is executed by the shell when the file is run as a command and the whole line (after the shebang) is passed to zasm as command line arguments. This is handy if lots of command line options must be passed to assemble this file.

This feature is also used in the regression selftest of zasm with command line option -T / --test.

Formerly any line starting with '#!' was ignored by the assembler.

Since version 4.3.6 line 1 is also examined by the assembler itself and options which modify the assembly are also applied by zasm as if they were given on the command line. This way they are also applied if the file is assembled normally and not run as a command. The options zasm looks for are:

```
    --z80, --z180, --8080
    --asm8080
    --dotnames, --reqcolon, --casefold, --flatops
    --ixcbr2, --ixcbxh
```

Note: there are also pseudo opcodes for these options,
namely .z80, .z180, .8080, .asm8080, .dotnames, and so on, which apply after they have been encountered by the assembler in pass 1, which may be a problem if they are defined too late.

Pseudo instructions

Pseudo instructions are instructions like 'org' or 'defs', which are written at the place of a Z80 assembler instruction but which are not Z80 instructions. Some pseudo instructions generate code, others don't.

Unless you use command line option '--reqcolon'Z80 and pseudo instructions must not start in column 1 but must always preceded by some spaces or tabs, else the assembler will mistakenly assume a label definition.

.area

```
         .area <name>
```

Switch to the code or data segment <name>. The following code will be stored in the segment <name>. You can switch between segments any time you like, appending some code here and appending some code there.

Equivalent to

```
     #code <name>
     #data <name>
```

except that it does not check the segment type. (data/code)

Example:

```
     ; define some uninitialized variables:
         .area _DATA
     _bu1::
         .ds 10
     _bu2::
         .ds 10

     ; define some initialized variables:
         .area _INITIALIZED
     _str3::
         .ds 10
     _a_counter::
         .ds 2

     ; define some executable code:
         .area _CODE
     _Intr:
         exx
         ex af,af'
         ld  hl,#_a_counter
         inc (hl)
         jr  NZ,00103$
         inc hl
         inc (hl)
         …
```

org, .org or .loc

```
         org <value>
         .org <value>
         .loc <value>
```

```
        org <value>, <fillbyte>
```

The pseudo instruction 'org' has two use cases:

It can define the start address of the assembler code or it can insert some space up to the new address.

Define the start address for the code

zasm requires that the source initially defines the address where the code will be stored and run. This can be done in either of two ways:

- Define a #target and subsequently some #code and #data segments
- Just define a code origin with 'org'.

Using 'org' is the traditional way for old sources. The code origin must be defined before any real code. When the origin is defined, zasm will silently create a single code segment. If you use 'org' you cannot specify a target or define multiple code segments or any data segments. Then

labels for data and variables in ram must be defined like other const values with 'equ' or similar. If you want to create code for a special target or if you want to include c sources then you must use #target, #code and #data and cannot use 'org' to define the start address of your code.

'org' actually defines two kinds of addresses:

- the 'physical' code address
- and the 'logical' code address

The logical code address is what you mostly work with. E.g. program labels and '$' refer to the logical code address. The logical code address can be moved with .phase and .dephase.

The physical code address is the address at which the generated code is actually visible to the cpu. (At least it should be.) Because most code is executed at the same address where it is visible to the cpu (which means: it is not copied somewhere else in ram before it is executed there) the physical and the logical address are most times the same. In case you need to access the physical address of the code at a position where you have shifted the logical address with .phase you can use '$$' instead of '$'.

Note: the physical code address is biased to the 'org' address resp. to the start address of the current #code or #data segment, not the start of the eprom!

Insert space up to a new address

The second use case for 'org' – which can also be used if you use #target, #code and #data – is to advance the code position up to the new logical origin, filling the space with the current segment's default fill byte. The default fill byte is $FF for ROMs and $00 for all other target formats. Since version 4.3.6 the fill byte can be specified as a second argument to 'org'. This is the same as with 'defs'.

Note: This second behavior of 'org' differs from the behavior in zasm v3. In zasm v3 'org' was solely used to set the logical origin. Use .phase and .dephase for this now!

The 'org' instruction is most commonly used to advance the code position up to the address of the next restart vector:

```
         org 8
```

is the same as

```
         defs    8-$
```

'org' behaves exactly the same if you use it in a #code or #data segment:

```
     #code _FOO, 0x4F00
         org 0x5000
         ...
```

This will insert 256 spaces at the beginning of the segment.

Name '.loc' is also recognized to support sources for different assemblers. org is recommended, the others are deprecated for new source.

defb, db, .db and .byte

```
     defb  <value> [,<value>...]
     db    <value> [,<value>...]
     .db   <value> [,<value>...]
     .byte <value> [,<value>...]
```

Store a series of bytes into the code.

The 'defb' pseudo instruction evaluates the argument expressions and puts them one byte after the other into the code segment.

Note: Due to the various methods for using 'defb' and 'defm' found in the wild zasm handles these two instructions in exactly the same way. This paragraph describes the 'byte oriented' usage cases. For the 'string oriented' usage cases see 'defm'.

```
         defb $123, _foo+10, (_N*2)+1, 'a', 'b'+1
```

Numeric literals can be written in various formats and the value can be a numeric expression.

The values are checked to be either a valid int8 or uint8 value, which means it must be in range -128 to +255.

If some bytes unexpectedly are reported to be out of range, then your source may assume no operator hierarchy. Then use command line option --flatops.

Character literals like 'a' are translated into the target system's character set if a #charset conversion has been defined.

There are some useful functions known by the expression evaluator:

- hi(N) and lo(N) to calculate the high and low byte of a 2-byte word

- min(N,N) and max(N,N) to calculate the smaller or the larger value of two values
- opcode(ld a,b) or similar to calculate the major opcode byte of a Z80 instruction
- defined(NAME) to determine whether a label is defined at this point in the source
- __line__ inserts the current line number

defw, dw, .dw, .word

```
defw  <value> [,<value>...]
dw    <value> [,<value>...]
.dw   <value> [,<value>...]
.word <value> [,<value>...]
```

This pseudo instruction inserts a single or a series of 2-byte words into the code. Words are stored with their least significant byte first. This means, $1234 is stored as $34, $12.

If you need them to be stored 'as seen' then use the 'defm' pseudo instruction.

.long

This pseudo instruction inserts a single or a series of 4-byte long words into the code. Long words are stored with their least significant byte first. This means, $12345678 is stored as $78, $56, $34, $12.

If you need them to be stored 'as seen' then use the 'defm' pseudo instruction.

defm, dm, .dm, .text and .ascii

```
defm   <text> [,<text>]
dm     <text> [,<text>]
.dm    <text> [,<text>]
.text  <text> [,<text>]
.ascii <text> [,<text>]
```

Store one or more text strings into the code segment.

Note: Due to the various methods for using 'defb' and 'defm' found in the wild zasm handles these two instructions in exactly the same way. This paragraph describes the 'string oriented' usage cases. For the 'byte oriented' usage cases see 'defb'.

The text string may be given in the following formats:

```
$12345678....   packed hex string
12345678h       packed hex string
'text'          character string, delimited by <'>, must not contain <'>
"text"          character string, delimited by <">, must not contain <">
```

The hex bytes or ascii bytes are stored in the same order as they appear in the string, e.g. $1234 is stored as $12,$34 and 'ab' is stored as 'a','b'.

The last byte of a character string or a packed hex string may be modified with an expression; commonly used is setting bit 7 of the last character of a message:

```
     defm   "this is my message"+$80
```

There are 4 predefined names for text strings:

```
     __date__        current date:      "Jan.31.96"
     __time__        current time:      "23:56:30"
     __file__        current file name  "filename.ext"
     __line__        current line numer "815"
```

Your assembler source is expected to be either 7-bit clean ascii or utf-8 encoded.

Characters in character strings are utf-8 decoded and then translated into the target system's character set if a #charset conversion has been defined. Characters must be representable in the target system's character set or an error will be flagged.

If no character set translation was defined, then the characters must, after utf-8 decoding, fit in a byte, e.g. 'ä' is allowed, '€' is not.

.asciz

```
     .asciz <text>
```

Store a 0-terminated character strings into the code segment.

The text string may be given in the following formats:

```
     'text'          character string, delimited by <'>, must not contain <'>
     "text"          character string, delimited by <">, must not contain <">
```

After the characters of the string an additional 0x00 byte is stored, which typically used in c sources as the string end marker.

Your assembler source is expected to be either 7-bit clean ascii or utf-8 encoded.

Characters in character strings are utf-8 decoded and then translated into the target system's character set if a #charset conversion has been defined. Characters must be representable in the target system's character set or an error will be flagged.

If no character set translation was defined, then the characters must, after utf-8 decoding, fit in a byte, e.g. 'ä' is allowed, '€' is not.

Note: some computers use a character set which contains character 0x00 as an ordinary character, most notably the ZX80 and ZX81. zxsp tries to detect this and will report an error if you try to use .asciz for these targets.

defs, ds, .ds, .block, .blkb and data

```
     defs   <value1> [,<value2>]
     ds     <value1> [,<value2>]
     .block <value1> [,<value2>]
     .blkb  <value1> [,<value2>]
     data   <value1>
```

This pseudo instruction inserts a block of <value1> bytes into the current code or data segment. The default fill pattern is $FF for ROMs and $00 for all other target formats. If you append a <value2>, this byte is used as pattern instead.

If the current segment is a data segment, then no fill pattern is allowed, as no actual code is stored.

Instruction 'data' is only allowed in a data segment and therefore it can never have a fill pattern assigned.

Multitiple names are recognized to support sources for different assemblers. defs or ds are recommended, the others are deprecated for new source.

align, .align

```
        align <value> [, <fillbyte>]
```

e.g.:

```
        align 8
```

Add fill bytes until the current logical address is a multiple of the given value.

Base for the alignment is the current logical code address, as accessible with '$'.

Positive addresses can be aligned to any value. If you happen to organize your addresses from -0x8000 to +0x7FFF then negative addresses can only be aligned to 2^N.

If only one argument is given, then 'align' inserts the default fill byte; 0xFF for target ROM and 0x00 for all others.

If a different fillbyte is required, then it can be appended as a second argument.

```
        align $100, 'E'|0x80
```

if, endif

```
    if <condition>
    ...
    endif
```

if' starts a block of assembler instructions, which is only assembled if the given <condition> is true. The <condition> must be evaluatable in pass 1. Conditional assembly may be nested. note: this may change.

The 'endif' instruction finishes conditional assembly as started with the 'if' instruction.

If the <condition> was false, then all instructions and assembler directives are ignored, except the matching 'endif' or nested 'if' … 'endif' ranges.

The assembler directives '#if', '#elif', '#else' and '#endif' are handled just like 'if' and 'endif'. note: this may change.

Normally the assembler directives with '#' should be used. Except that 'if' and 'endif' can occur in macros and the expanded macro can conditionally exclude some code.

end, .end

Define the logical end of your source.

end must not be inside an #if branch or in a #local scope. This is so that using end does not disable the test for a missing #endif or #endlocal.

Anything after 'end' is ignored by the assembler.

This pseudo instruction is an alias for the assembler directive #end.

This instruction is optional.

include

Use of the 'include' pseudo instruction is an alternate syntax for the '#include' assembler directive.

See chapter '#include' for details.

incbin

```
        incbin <filename>
```

Inserts a binary file. The file name must be given as an absolute path or relatively to the current source file.

The file's contents are just copied into the code segment verbatim.

Examples:

```
        incbin "image.gif"      ; file in same directory
        incbin "./image.gif"    ; file in same directory
        incbin "pics/g1.gif"    ; file in subdirectory "pics/"
```

This pseudo instruction does the same as the assembler directive #insert:

```
    #insert "image.gif"         ; file in same directory
```

.globl

```
        .globl
```

For the sake of sdcc:

Declare the named label to be global.

The label is either defined in the current local scope after this declaration or it is referenced there.

Undefined global labels can be picked up later by the '#include library' assembler directive.

Define a global label

```
    #local
            .globl foo
            ...
    foo:    nop
            ...
```

```
#endlocal
```

The same effect can be achieved by putting two colons after the label definition:

```
#local
        ...
foo::   nop
        ...

#endlocal
```

Refer to a global label

```
#local
        .globl foo
        ...
        jp  foo
        ...
#endlocal
```

Unless you have a nested local context, this is identically to:

```
#local
        ...
        jp  foo
        ...
#endlocal
```

In almost all cases the .globl declaration is superfluous. zasm automatically adds the label to the list of undefined local labels and pushes it at each #endlocal to the outer context until it finds a context where it is defined – finally the global context where it can be picked up by the '#include library' assembler directive if it is still undefined then.

Actually all references to symbols in system libraries as included by the c compiler are handled this way, without the need to define them all ahead.

.globl is useful in nested local contexts where the same name might be defined in an outer but not the global context.

.globl may also be useful if you want to redefine a redefinable label. Inside a local context it is impossible to redefine a label, because this would define a new local label instead, except if you define it .globl.

```
foo     defl    0
        ...
#local
        ...
        .globl  foo
foo     defl    1
        ...
#endlocal
```

#local, #endlocal, .local and .endlocal

This defines a local scope for normal label definitions. This way you can safely use 'standard' names for jump targets in included files without worrying about name collissions. You can still push labels to the global scope though.

#local contexts can be nested.

Example: Wrap an included source file to avoid label name conflicts.

Note: zasm does this automatically for c files, but not for assembler files:

```
#local
#include "util/u.asm"
#endlocal
```

Define global variables from inside a #local context:

```
#local
boo::   ld a,0      ; '::' makes 'boo' global
        .globl fax
fax:    ret         ; 'fax' is now also global
#define bar 4711    ; labels defined with #define are also global.
#endlocal
```

Note: used but locally undefined labels are automatically pushed to the surrounding scope by #endlocal and will finally reach the global scope, if they were not defined in an intermediate surrounding #local scope. There they can be picked up and resolved by the #include library directive.

The .globl pseudo instruction can also be used to make sure that a certain label is global and not accidentially defined in a surrounding #local scope.

In some cases you need labels to be resolved in pass 1, e.g. for #IF <cond> or for .rept. Then you run into problems when you refer to global labels from within a #local scope: zasm does not know for shure, whether it should use the valid global label, because a local label with this name still might be defined after this point of use. Therefore all references to global labels from within a local scope are never valid in pass 1. To solve this problem you can declare this label global using the .global pseudo instruction.

macro, .macro, endm and .endm

'.macro' starts a macro definition and '.endm' ends it. Macros are assigned a name by which they can later be used at the place of an ordinary instruction to insert the macro body.

There are quite numerous formats used by different assemblers and zasm tries to support at least some of them out of the box. In the following code examples the instruction names with or without a dot can be used interchangeably.

Define a simple macro without arguments

```
<name>  .
macro
        <instr>
        ...
        .
endm
```

Alternate syntax:

```
        .macro <name>
        <instr>
        ...
        .endm
```

Example:

```
counter = 0
    .macro COUNT
counter = counter + 1
    .endm

    ...
    COUNT
    ...

final_count equ counter
```

This defines a counter for something, e.g. Forth words if your source happens to be a Forth interpreter. It uses a redefinable label which is redefined every time the macro is used in the source. The final value of the redefinable label is only really the final value at the end of your source. In order to use it before the last occurance of the macro, it is finally copied into another label named 'final_count' which can be used everywhere in the source.

Define a macro with arguments

```
<name>  .macro <tag><name1> [ , <tag><name2> … ]
        <instr>
        ...
        .endm
```

Alternate syntax:

```
.macro <name> <tag><name1> [ , <tag><name2> … ]
<instr>
...
.endm
```

Invocation:

```
<name> <some text> [ , <some text> … ]
```

This defines a macro with some arguments. Either of both definition styles are possible. Additionally there are different methods for defining and referencing the symbol arguments:

```
foo:    .macro A, B         ; 1)
        ld  &A, &B
        .endm

        .macro foo, A, B    ; 2)
        ld  \A, \B
        .endm

foo:    .macro &A, &B       ; 3)
        ld  &A, &B
        .endm

        .macro foo, %A, %B  ; 4)
```

```
                ld  %A, %B
                .endm
```

Argument processing in the macro definition

A special character, a 'tag', should be preprended to the argument names in the head and body. This 'tag' varies from assembler to assembler. zasm treats any non-letter character at start of the argument names as this 'tag'. Most commonly used is '&'.

Some assemblers required the tag only in the macro body, not in the head, see version 1) and 2) in the above example. zasm supports this "as seen in source examples" in the following way:

- If the macro is defined as <name> macro <arguments…> then the tag is '&'.
- If the macro is defined as macro <name> <arguments…> then the tag is '\'.

Also, the tag is not required in the body if it was not written in the argument list. Then argument references must be full words, whereas with tag they are also detected as part of a words.

The use of '&' is recommended. All other tags are to support various sources out of the box only!

Example:

```
foo:    macro &A
        ld  b,&A
foo&A:  ld  a,' '       ; <-- simple unique label name
        rst 8
        djnz    foo&A
        endm

        ...
        foo 20
        ...
        foo 8
```

In this example the argument is used as the second argument for 'ld b,N' and to construct a unique label name for each invocation. Another solution would have been to use a counter and a redefinable label.

```
.macro foo_define &NAME, &rr
        ld_&NAME_&rr = ld_int_&rr
.endm
```

In this example '&NAME' in 'ld_&NAME_&rr' is also replaced, though 'NAME' is only the first 4 characters of the potential 5-letter name 'NAME_'.

An example without tag character:

```
DWA:  MACRO WHERE
      DB    (WHERE >> 8) + 128
      DB    WHERE & 0FFH
      ENDM
```

Argument processing in the macro invocation

There are two methods for argument parsing, the standard 'clean & simple' one, and one for complex arguments.

Standard arguments

Arguments must follow the basic syntax rules: strings or character literals must be closed (balanced delimiters) and arguments end at the next comma, semicolon or line end. Brackets are not required to match.

```
foo a, 1+2, a*(1+2)
foo (, "hi, i'm Kio!", )    ; 3 arguments
```

Complex arguments

Complex arguments start with an opening '<' and run up to the next '>' which is followed by a comma, a semicolon or the line end. With the help of the angle brackets you can pass unbalanced ''' and '"' or a comma or a semicolon as one argument to a macro. It is also helpful if you need to pass spaces at the start or end of an argument, or even just a space.

```
foo <,>, <;>, <">           ; 3 arguments: , ; and "
foo <<>, < >, <ld a,b>      ; 3 arguments: < space and ld a,b
foo >, ><,                  ; 3 arguments: > >< and nothing
```

There are still some impossible combinations, e.g. it is not possible to pass <>, (3 characters) as an argument.

'{...}' replacement

Since version 4.4.6 numeric expressions between '{' and '}' are evaluated and immediately replaced with the resulting text. This also happens for macro arguments. If you mix this with the complex argument style then you can probably run into interesting problems. B-)

Labels in macros

Frequently you need some program labels in a macro, but if you use a macro more than once then you'll have multiple definitions of this label name and assembly fails.

- Use redefinable labels instead which are defined with 'defl', '=' or ' set'
- or use a local context between '.   local' and '.   endlocal' (since version 4.4.6)
- or construct unique label names using a redefinable counter and text replacement with '{…}'.

Literal text replacement of expressions between '{' and '}' Expressions between '{' and '}' are evaluated and immediately replaced with the result

- Since version 4.2.6 in macros
- Since version 4.4.6 everywhere

Example:

```
cnt:    .equ  0         ; initialize redefinable label 'cnt'

; calculate unsigned max
; &A = umax(&A,&B)
; usable for a and hl.

umax:   .macro &A, &B
        and   a
        sbc   &A,&B
        jr    nc,L{cnt}
        ld    &A,&B
        jr    L{cnt+1}
L{cnt}: add   &A,&B
L{cnt+1}:
cnt     .equ  cnt+2     ; next macro call will use L2 and L3 and so on
        .endm
```

Difference between version 4.2.6 and 4.4.6:

Since version 4.2.6 the text replacement was done at the point of the macro replacement. This had a subtle side effect: All expressions between '{' and '}' were already evaluated before the lines from the macro were assembled. Therefore redefinitions of labels performed inside the macro had no effect on '{…}' expressions until the end of the macro.

So in the above example the line cnt .equ cnt+2 had no effect on the expressions between '{' and '}' even if this line was the very first in the macro; but it would have effected all other normal expressions in the macro as expected. (in the above example there is none except in the cnt incrementing expression itself.)

```
cnt = 0
.macro Foo
cnt = cnt+1
L{cnt}: .dw cnt ; 1)
cnt = cnt+1
L{cnt}: .dw cnt ; 2)
.macro
```

In this example the label in line 1) would still be 'L0' while the value stored with '.dw' would be 1.

In line 2) the label again would still be 'L0' (and assembly fails) while the value stored with '.dw' would be 2. This was irritating and is no longer the case since version 4.4.6.

Since version 4.4.6 text replacements between '{' and '}' are now possible everywhere in the source and they are done when the lines are actually assembled. Now in the above example the label names are 'L1' and 'L2' as expected.

conditional assembly with '.if' in macros

It is possible to conditionally exclude portions of the macro by use of the pseudo instructions '.if', '.else' and '.endif'. (do not use '#if' and '#endif'!)

Assembler directives in macros

Assembler directives (starting with '#') inside macros are possible but deprecated.

Also, you should not use '#' for your argument tag.

Nested macro definitions

Since version 4.4.6 macro dfinitions inside macro definitions are possible and may make some sense with text replacement between '{' and '}'. But you really should know what you are doing!

Not implemented:

- Reference to arguments without a tag character
- Keyword 'local' before a <NAME> to define a single local label: try to use 'defl'. since version 4.4.6: use a local context: '.local' … '.endlocal'

rept, .rept, endm and .endm

```
rept    <value>
    <instr>
    ...
endm
```

This pseudo instructions defines a sequence of source which shall be stored into code multiple times. By use of a redefinable label the stored values may even be adjusted slightly. This is a lightweight form of an ad-hoc macro.

The names with or without dot may be used interchangeable.

Assembler directives (starting with '#') are not allowed inside macros.

Starting with version 4.1.4 the names "DUP" and "EDUP" are also accepted for "REPT" and "ENDM".

Example:

```
rept    $2000 - $
    defb    0
endm
```

Which does the same thing as 'defs $2000-$' or even 'org $2000'. Just more complicated.

```
v   defl    0
    rept    8
    lda     regs2 + v/2
v   defl    v+2
    cpi     v
    jnz     0
    endm
```

This seems to be 8080 assembler source. B-)

dup, .dup, edup and .edup

```
    dup <value>
    <instr>
    ...
    edup
```

DUP and EDUP were introduced as an alias for REPT and ENDM in version 4.1.4.

This pseudo instructions defines a sequence of source which shall be stored into code multiple times. By use of a redefinable label the stored values may even be adjusted slightly. This is a lightweight form of an ad-hoc macro.

The names with or without dot may be used interchangeable.

Assembler directives (starting with '#') are not allowed inside macros.

Examples:

```
; from the source of dzx7_lom:
    DUP     3
    ldi
    add     a
    jr      c, dzx7l_process_ref
    EDUP


v   defl    0
    DUP     8
    ld      a, regs2 + v/2
v   defl    v+2
    cp      a, v
    jr      nz, 0
    EDUP
```

.phase and .dephase

```
.phase <address>
...
.dephase
```

This pseudo instruction sets the logical origin for the following code to the new address. The new logical origin is valid until the .dephase instruction resets it to the physical origin or another .phase instruction sets it to another value. (note: in zasm v3 'org' was used to change the logical origin.)

The logical code address is what you mostly work with. E.g. program labels and '$' refer to the logical code address. This logical code address can be moved with .phase and .dephase.

The physical code address is the address at which the code stored by the assembler is supposed to be visible to the cpu. (At least it should be.) Because most code is executed where it was stored (which means: it was not copied somewhere else in ram before it is executed there) the physical and the logical address are most times the same.

.phase and .dephase are typically used if some portions of code are copied from rom into ram, where it can be modified by the running program.

In case you need to access the physical address of the code at a position where you have shifted the logical address with .phase you can use '$$' instead of '$'.

Note: the physical code address is biased to the 'org' address resp. to the start address of the current #code or #data segment, not the start of the eprom!

Example: relocated jump table

In this example a jump table is copied from rom to ram, either to assert fixed addresses or to allow the running program to patch some entries:

```
            ...                 ; some code
            ccf
            ret
            .phase  $FE00       ; shift origin
rom_table:  equ $$
ram_table:  equ $

putchar:    jp      putchar_rom
puttext:    jp      puttext_rom
getchar:    jp      getchar_rom
getline:    jp      getline_rom

table_size: equ     $ - ram_table

            .dephase            ; back to physical code address

gizmo:      ld      de,ram_table
            ld      hl,rom_table
            ld      bc,table_size
            ldir

            ld  a,0             ; more code
            ...
```

#assert

```
#assert <condition>
    .assert <condition>
```

Assert that the given condition is true.

This directive might be used to detect logical or overflow errors in your source. It is used by zasm itself to assert that loaded library files actually defined the label they promissed to define. It is also used in zasm's assembler test suite.

Example:

```
#assert ( $ & 7 ) == 0      ; assert that current code position
                            ; is a multiple of 8

.area GSINIT
.
assert DATA_size > 1       ; else ldir will run amok

ld  bc,DATA_size - 1
ld  de,DATA + 1
ld  hl,DATA
ld  (hl),0
ldir
```

.z80, .z180 and .8080

```
    .z180
    .org 0
    ...
```

These pseudo instructions declare the type of processor and .8080 also the assembler syntax in use. They are related to the command line options --z80, --z180, --8080 and --asm8080. These instructions can be used to replace the command line options. They must occur befor the first 'org' instruction or segment definition.

These pseudo instructions will fail if another cpu was already selected on the command line.

Note: the behavior slightly changed in version 4.3.4.

Target cpu z80

```
        .z80
```

Declare the source to be for a Z80. This is the default, so it is normally not required.

Target cpu z180

```
        .z180
```

Declare the source to be for a Z180 or HD64180.

The additional instructions are enabled and all illegal instructions are disabled.

Target Intel 8080

```
        .8080
```

Use 8080 assembler syntax and declare the source to be for a Intel 8080 cpu.

This is more like command line option '--asm8080' because it also selects 8080 assembler syntax.

In 8080 assembler, label names are not case sensitive, it implies '--casefold'.

Commands for command line options

The following instructions must all occur befor the first org or segment definition and they do the same as the similar named command line option. You may put these pseudo ops at the start of your assembler source to eliminate the need to pass the command line options every time you assemble it.

*--ixcbxh, .ixcbxh, _ixcbxh_*
*--ixcbr2, .ixcbr2, _ixcbr2_*

Related command line option, pseudo instruction and predefined label

IXCB is used here as a mnenonic for the prefix IX or IY plus prefix 0xCB instructions like 'rr', 'bit', 'set' and 'res'. These are allowed for those opcodes which use the '(hl)' register only. But people have found out what they do when you use one of the other, regular registers: They may additionally copy the result into the selected register or they may, if you use register 'h' or 'l', access the upper or lower byte of the index register.

What a CPU does depends on it's hardware. Original Zilog Z80 CPUs should behave as for --ixcbr2. Others allow the access to the index register halves, some just behave as for '(hl)' for all other registers as well and some, like the Z180, even trap all illegal instruction.

Option --ixcbr2 enables the 'second register target' opcodes like

```
	set 0,(ix+0),b
```

and option --ixcbxh enables the 'index register halves' opcodes like

```
	set 0,xh
```

Obviously they are mutually exclusive.

If command line option --ixcbr2 or pseudo opcode .ixcbr2 was used, then the label _ixcbr2_ is defined and can be tested with 'defined(_ixcbr2_)' in expressions.

If command line option --ixcbxh or pseudo opcode .ixcbxh was used, then thelabel _ixcbxh_ is defined and can be tested with 'defined(_ixcbxh_)' in expressions.

--dotnames, .dotnames

Related command line option, pseudo instruction and predefined label

Allow label names to start with a dot.

First dots were not allowed in label names. Then some assemblers prepended a dot to the pseudo instructions to easily distinguish them from normal instructions and label names, so they could start at any column of a source line. Having seen that, some people imme diately developed the need to start their label names with a dot as well…

Normally, without this option, only pseudo instructions like '.ds' or '.org' can start with a dot. Then they may be written in any column of a source line regardless of the '--reqcolon' setting.

With this setting the limitations described at '--reqcolons' also applies to pseudo instructions starting with a dot because the dot no longer makes them special and implicitely a non-label.

The following example is only valid source with --dotnames and --reqcolon:

```
.org    0               ; starts in column 0 => requires --reqcolon
    .ds 0x66            ; one of the original dot named pseudo instructions
    jp  .nmi_handler    ; requires --dotname
```

Of course you could clean up your source as well…

Prior to version 4.3.4 a label _dotnames_ was also defined.

--reqcolon, .reqcolon

Related command line option, pseudo instruction and predefined label

Select syntax variant where colons ':' are required after program label definitions.

Normally colons are not required and the assembler decides whether a name is a label definition or something else by the position where it starts: label names must start in column 1 and instructions must be preceded by some space.

This example must not be compiled with --reqcolon or --dotnames:

```
.org 0                  ; due to the dot this pseudo instruction is
                        ; allowed in column 1
foo     equ 0xF00       ; label definition
```

```
bar:    and a           ;
program label def and a instruction
        jp  foo         ; instruction
shelf   db  0           ; program label def and a pseudo instruction
```

But some source files don't obey this widely accepted rule and put instructions in column 1 or – actually seen – start label definitions indented with some tabs.

As long as only pseudo instructions starting with a dot are put in column 1 and option --dotnames is not required, this is no problem. In all other cases you need this option --reqcolon and, of course, then colons are required after program label definitions, but still not after other label definitions with 'equ' and the like:

This example must be compiled with --reqcolon:

```
; funny indented:
org 0                       ; 'org' would be recognized as a label
                            ; without --reqcolon
    foo     equ 0xF00       ; label definition
    bar:    and a           ; program label def and a instruction
jp  foo                     ; instruction
    shelf:                  ; program label definition
.db 0                       ; a pseudo instruction
```

Prior to version 4.3.4 a label _reqcolon_ was also defined.

--casefold, .casefold

Related command line option, pseudo instruction and predefined label

Tell the assembler that label names are case insensitive and that the source does not distinguish between uppercase and lowercase writing of label names.

Note: instruction and register names are always case insensitive.

This option is implied by '--asm8080'.

Normally label names are case sensitive or people just tend to just write them allways the same.

Prior to version 4.3.4 a label _casefold_ was also defined.

--flatops, .flatops

Related command line option, pseudo instruction and predefined label

Evaluate expressions strictly from left to right, disobeying any operator precedence.

Old assemblers sometimes didn't know about operator precedence and were designed this way for simplicity.

Use this option if error messages about byte values or offsets out of range are reported by zasm and these expressions make more sense when evaluated this way.

Normal operator precedence in zasm is:

```
unary ! ~ + -  ▶  >> <<  ▶  & | ^  ▶  * / %  ▶  + -  ▶  > >= etc.  ▶  && ||
▶  ?:
```

Note that this is not the precedence as defined for C. B-). The hierarchy of operators in c is very bad.

Prior to version 4.3.4 a label _flatops_ was also defined.

equ

Named values:

```
<name>[:][:] equ <value>
```

Frequently you start your source with a buch of definitions for const values, like printing control codes or port addresses. These definitions require a keyword after the label, in exchange a colon is never required, but may be present.

```
foo:    equ 255 -1
bar     equ 255 -2
shelf  equ 0xffff
#define foobar foo & bar
```

You can append two colons '::' to the label name if you define a label inside a local context (after #local) but want to make it globally visible:

```
#local
        ...
foo::   equ $+2
        ...
#endlocal
```

Additionally zasm supports the use of the c preprocessor-style definition using '#define'. Labels defined with '#define' are always global.

For more information see the chapter about label definition.

defl, set and '='

Define a redefinable named value:

```
<name>[:][:] set  <value>
<name>[:][:] defl <value>
<name>[:][:] = <value>
```

Labels defined with this keyword can be redefined at some later position.

Especially for use in macros it is useful to allow labels which can be redefined to another value.

Example: define a macro which counts the occurances af something:

Note: the final count is only valid after the last use of this macro in your source!

```
foo     set 0
        macro COUNT
foo     set foo+1
        endm
```

Example: use of a redefinable label in place of a program label:

Note: forward jumping couldn't be implemented this way!

```
        macro WAIT
        ld      b,10
foo     defl    $
        call    wait_1ms
        djnz    foo
        endm
```

The label's value is valid from the position where it was defined to the position where it is redefinedor the end of the source. References to this label before it's first definition are illegal!

For more information see the chapter about label definition.

Label definition

```
<name>[:][:] [<instruction>]
<name>[:][:] equ  <value>
<name>[:][:] =    <value>
<name>[:][:] set  <value>
<name>[:][:] defl <value>
#define <name>    <value>
```

Label names may be of any length. Upper or lower case is distinguished. The label name may be followed by one or two colons ':'.

Normally, label definitions must start in column 1 and instructions must be indented with some spaces. The assembler decides based on the starting column of a name whether it is a label definition or an instruction.

Any name which starts in column 1 and is not an assembler directive (which always starts with a hash '#') or a comment (which always starts with a semicolon ';') is treated as a label definition.

unless -- dotnames is also set,

pseudo instructions starting with a dot are also recognized in column 1.

--reqcolon

If the source is assembled with command line option '--reqcolon', then one colon is required after the label name. This allows label definitions to start in any column, either in column 1 as usual or indented with some spaces and tabs, and it allows instructions to start in column 1. The assembler decides based on the existance or absence of a colon whether it is a label definition or an instruction.

Note: even with --reqcolon no colon is required after labels defined with 'equ' and similar.

Recommendation: Start label definitions in column 1, indent instructions with one or two tabs and put a colon after label definitions.

--dotnames

Normally label names cannot start with a dot. Names starting with a dot were introduced by some assemblers to easily distinguish betwen label names, instructions and assembler pseudo instructions, so these could be written in column 1 if the programmer desired so. Having seen that, some people immediately developed the need to start their label names with a dot as well…

Recommendation: Don't start label names with a dot.

--casefold

Normally label names are case sensitive or people just tend to just write them allways in the same way. If an older source does not distinguish between uppercase and lowercase writing of label names, then you can use this option to tell zasm to ignore upper and lower case as well.

Note: instruction and register names are always case insensitive.

This option is implied by --asm8080.

Recommendation: Don't use this option unless required.

Types of labels

Program labels

```
<name>[:][:] [<instruction>]
```

Program labels basically are what we just talked about: A name put in front of an instruction to give the current code position a name so that it can be used in a later call or jp statement.

```
foo:    jp  _bar
```

This defines a program label named 'foo' and the instruction jumps to a label named '_bar' which is likely a program label as well.

The address of the current instruction can also be referenced as '$', so in some easy cases you don't need to define a label.

If you request that your list file includes accumulated cpu cycles, then the counter is reset at every program label definition.

Named values

```
<name>[:][:] equ  <value>
<name>[:][:] =    <value>
#define <name>    <value>
```

Frequently you start your source with a bunch of definitions for constant values, like printing control codes or port addresses. These definitions require the keyword 'equ' after the label, in exchange a colon after the label name is never required, even with --reqcolon, but may be present.

```
foo     equ 255 -1
bar     equ 255 -2
shelf   =   0xffff
#define foobar foo & bar
```

Most assemblers require keyword 'equ', some use '='. The use of 'equ' is recommended.

Additionally zasm supports the use of the c preprocessor-style definition using '#define'.

Redefinable named values

```
<name>[:][:] set  <value>
<name>[:][:] defl <value>
```

Especially for use in macros it is possible to define labels which can be redefined to another value. These definitions require the keyword 'set' or 'defl' after the label, in exchange a colon after the label name is never required, even with --reqcolon, but may be present.

Note: don't confuse the 'set' pseudo instruction with the Z80's 'set' instruction!

This defines a macro which counts the occurances of something:

Note: the final count is only valid after the last use of this macro in your source!

```
foo     set 0
        macro COUNT
foo     set foo+1

endm
```

Use of a redefinable label in place of a program label:

Note: forward jumping can't be implemented this way!

```
        macro WAIT &N
        ld      b,&N
foo     defl    $
        call    wait_1ms
        djnz    foo
        endm
```

The value is valid from the position where it was defined to the position where it is redefined or the end of the source. References to this label before it's first definition are illegal!

global labels

Labels are defined in the current 'scope', which is either the default global scope or, after #local, a local scope.

Normally labels defined after #local are only known in the local scope:

```
#local
foo:    nop
        ...
        jr  foo
#endlocal
```

You can also define global labels inside a local scope:

Just put two colons '::' after the label name or use the pseudo instruction '.globl':

```
#local
        .globl  bar
foo::   nop             ; global
        ...
bar:    nop             ; global
        ...
shelf:: equ foo+1       ; global
#endlocal
```

Reusable labels

zasm supports the 'recyclable' labels used by sdcc:

```
foo::
        ...
123$:   nop
        jp  123$
        jp  199$
199$:   nop
        ...
bar::
        ...
123$:   nop
        ...
```

The name of a reusable label is only valid in the range between two normal program labels.

The naming scheme is a little but unlucky.

If you put a dollar sign '$' after a number it becomes a label name. :-/

Literal text replacement using "{...}"

since version 4.2.6: in macros

since version 4.4.6: everywhere

Before a source line is assembled, expressions between "{" and "}" are resolved.

Literal text replacement is typically used to create calculated label names and can solve a common problem with macros.

```
mx = 0  ; a counter which serves as a seed for labels in all macros
;
.macro extend_sign  ; a macro with a forward jump
    ld  h,0
    ld  a,l
    add a
    jr  nc,L{mx}    ; jump forward
    dec h
L{mx}:
mx = mx+1           ; next use of macro will use next number
.endm
```

While backward jumps in labels can simply use a redefinable label to jump to, this is not possible with forward jumps. Besides other solutions (you could create a .local context in your macro) you can generate label names using a counter. In the above example the first use of the macro would use label L0, the second L1 and so on.

You can also iterate over a range of programmatically named labels:

```
_n = 0
.rept mx
    .dw L{_n}
_n = _n +1
.endm
```

This will generate an array with all addresses where label L{mx} in the above example was used.

Actually you can implement arrays of labels this way e.g. to automatically create jump tables.

CPU instructions

8080 instructions

These are the instructions which are common to the Z80 and the Intel 8080.

The 8080 lacks the index registers, the second register set and has no jump relative instructions. Also it has no 0xCB and 0xED prefixed commands.

Note: for the 8080 assembler mnemonics see chapter '8080 assembler instructions'.

```
nop
ld RR,NN        RR = BC DE HL SP
add hl,RR
inc RR
dec RR

ld (bc),a
ld a,(bc)
ld (de),a
ld a,(de)
ld (NN),hl
ld hl,(NN)
ld (NN),a
ld a,(NN)

inc R           R = B C D E H L (HL) A
dec R
ld R,N

rlca
rrca
rla
rra
daa
cpl
scf
ccf

halt

ld  R,R         R = B C D E H L (HL) A
                except ld (hl),(hl)

add a,R         R = B C D E H L (HL) A
adc a,R
sub a,R
sbc a,R
and a,R
xor a,R
or  a,R
cp  a,R

ret
ret CC
jp  NN
jp  CC,NN
call NN
call CC,NN
rst  N

pop  RR         RR = BC DE HL AF
push RR         RR = BC DE HL AF
```

```
add a,N
adc a,N
sub a,N
sbc a,N
and a,N
xor a,N
or  a,N
cp  a,N

out (N),a
in a,(N)

ex (sp),hl
ex de,hl
di
ld sp,hl
ei
```

Z80 instructions

List of all Z80 opcodes, which were not already present in the Intel 8080. These are all instructions which use the index registers and the second register set, the relative jumps and the 0xCB and 0xED prefixed opcodes.

```
ex af,af'
exx
djnz DIS
jr  DIS
jr  nz,DIS
jr  z,DIS
jr  nc,DIS
jr  c,DIS

rlc R           R = B C D E H L (HL) A
rrc R
rl  R
rr  R
sla R
sra R
srl R

bit N,R
res N,R

set N,R

in  R,(c)       R = B C D E H L A
out (c),R

in  f,(c)

sbc hl,RR       RR = BC DE HL SP
adc hl,RR
ld  (NN),RR
ld  RR,(NN)

neg
retn
im  N           N = 0 1 2
ld  i,a
ld  r,a
ld  a,i
ld  a,r
reti
rrd
```

```
rld

ldi
cpi
ini
outi
ldd
cpd
ind
outd
ldir
cpir
inir
otir
lddr
cpdr
indr
otdr

ld  RR,NN       RR = IX IY
add hl,RR
inc RR
dec RR
ld  (NN),RR
ld  RR,(NN)
pop  RR
push RR
ex  (sp),RR
ld  sp,RR

inc R           R = (IX+N) (IY+N)
dec R
ld  R,N
add a,R
adc a,R
sub a,R
sbc a,R
and a,R
xor a,R
or  a,R
cp  a,R

ld  R1,R2       R1 = B C D E H L A  and  R2 = (IX+N) (IY+N)
ld  R1,R2       R1 = (IX+N) (IY+N)  and  R2 = B C D E H L A
```

Z180 instructions

These are the instructions added in the Z180 / HD64180 cpu.

```
in0  R,(N)      R = B C D E H L F A
mult RR         RR = BC DE HL SP
out0 (N),R      R = B C D E H L A
otim
otdm
otimr
otdmr
slp
tst  R          R = B C D E H L (HL) A
tst  N
tstio N
```

Illegal instructions

List of all illegal Z80 opcodes. The Z80 cpu does not trap undefined opcodes but 'just does something' instead. For many undocumented opcodes it is well known what they do, and sometimes it is something useful.

Undocumented opcodes after a combination of index register plus prefix 0xCB behave differently on different CPU brands.

On CPUs like the Z180 / HD64180 which trap illegal opcodes these instructions cannot be used.

```
sll R               R = b c d e h l (hl) a

out (c),0xFF        for NMOS CPUs
out (c),0           for CMOS CPUs
in  f,(c)
in  (c)             syntax variant

inc R               R = xh, xl, yh, yl or syntax variant: ixh, ixl, iyh, iyl
dec R
ld  R,N
add a,R
adc a,R
sub a,R
sbc a,R
and a,R
xor a,R
or  a,R
cp  a,R

ld  R1,R2           R1 = b c d e xh xl a  and  R2 = xh or xl
ld  R1,R2           R1 = b c d e yh yl a  and  R2 = yh or yl
ld  R1,R2           R1 = xh or xl  and  R2 = b c d e xh xl a
ld  R1,R2           R1 = yh or yl  and  R2 = b c d e yh yl a

--


ixcbr2:

rlc (RR+dis),R      RR = ix iy,  R = b c d e h l a
rrc (RR+dis),R
rl  (RR+dis),R
rr  (RR+dis),R
sla (RR+dis),R
sra (RR+dis),R
sll (RR+dis),R
srl (RR+dis),R

bit N,(RR+dis),R    RR = ix iy,  R = b c d e h l a
res N,(RR+dis),R

set N,(RR+dis),R

--


ixcbxh:

rlc R               R = xh xl yh yl or ixh, ixl, iyh, iyl
rrc R
rl  R
rr  R
sla R
sra R
```

```
srl R

bit N,R             R = xh xl yh yl or ixh, ixl, iyh, iyl
res N,R

set N,R
```

Syntax variants

```
zasm supports different syntax for some opcodes:
ex  hl,de       ex  de,hl
ex  (sp),RR     ex  RR,(sp)     RR = hl, ix, iy
jp  (RR)        jp  RR
in  R,(c)       in  R,(bc)
out (c),R       out (bc),R      R = b c d e h l (hl) a
in  a,(N)       in a,N
out (N),a       out N,A
rst 0           rst 0
rst 1           rst 8
rst 2           rst 16
rst 3           rst 24
rst 4           rst 32
rst 5           rst 40
rst 6           rst 48
rst 7           rst 56
add a,R         add R           R = b c d e h l (hl) a
adc a,R         adc R               and (ix+dis) (iy+dis)
sub a,R         sub R
sbc a,R         sbc R
and a,R         and R
xor a,R         xor R
or  a,R         or  R
cp  a,R         cp  R
```

Some variants apply to notation for arguments:

```
ld  a,N         ld  a,#N
ld  a,(ix+0)    ld  a,(ix)
ld  a,(ix+dis)  ld  a,dis(ix)
```

Beyond that, zasm provides convenience definitions for compound instructions to increase readability and maintainability by reducing the number of lines in a source files.

Convenience definitions

These are convenience definitions for combinations of real instructions.

Those which are made from illegal opcodes can't be used for the Z180.

All opcodes which do not use the index registers or the 0xCB group can also be used for the 8080.

All these combinations have no side effect.

```
ld  bc,de
ld  bc,hl
ld  de,bc
ld  de,hl
ld  hl,bc
ld  hl,de

ld  bc,ix       ; illegal ...
ld  bc,iy       ; ...
ld  de,ix       ; ...
ld  de,iy       ; ...
ld  ix,bc       ; ...
ld  ix,de       ; ...
ld  iy,bc       ; ...
ld  iy,de       ; ...

ld  bc,(ix+dis)
ld  bc,(iy+dis)
ld  de,(ix+dis)
ld  de,(iy+dis)
ld  hl,(ix+dis)
ld  hl,(iy+dis)

ld  (ix+dis),bc
ld  (iy+dis),bc
ld  (ix+dis),de
ld  (iy+dis),de
ld  (ix+dis),hl
ld  (iy+dis),hl

ld  bc,(hl)
ld  de,(hl)
ld  bc,(hl++)
ld  de,(hl++)
ld  bc,(--hl)
ld  de,(--hl)

ld  (hl),bc
ld  (hl),de
ld  (--hl),bc
ld  (--hl),de
ld  (hl++),bc
ld  (hl++),de

ld  (--bc),a
ld  (--de),a
ld  (bc++),a
ld  (de++),a
ld  a,(--bc)
ld  a,(--de)
ld  a,(bc++)
ld  a,(de++)

ld  R,(hl++)
ld  R,(--hl)

rr  bc              ; 0xCB group
rr  de
rr  hl
sra bc
sra de
sra hl
srl bc
srl de
srl hl
rl  bc
rl  de
rl  hl
sla bc
sla de
sla hl
sll bc              ; sll undocumented
sll de              ; sll undocumented
sll hl              ; sll undocumented

rr  (hl++)          ; 0xCB group
rrc (hl++)
```

```
     rl  (hl++)
     rlc (hl++)
     sla (hl++)
     sra (hl++)
     sll (hl++)          ; sll undocumented
     srl (hl++)
     bit N,(hl++)

     set N,(hl++)
     res N,(hl++)

     rr  (--hl)          ; 0xCB group
     rrc (--hl)
     rl  (--hl)
     rlc (--hl)
     sla (--hl)
     sra (--hl)
     sll (--hl)          ; sll undocumented
     srl (--hl)
     bit N,(--hl)

     set N,(--hl)
     res N,(--hl)

     add (hl++)
     adc (hl++)
     sub (hl++)
     sbc (hl++)
     and (hl++)
     or  (hl++)
     xor (hl++)
     cp  (hl++)

     add (--hl)
     adc (--hl)
     sub (--hl)
     sbc (--hl)
     and (--hl)
     or  (--hl)
     xor (--hl)
     cp  (--hl)
```

Load/store Quad Registers

since version 4.4.2.

actually, these are not implemented! only usable in '.expect' so far.

Quad registers are combinations of two 16 bit registers BC, DE, HL, SP, IX or IY.

these can also be .expected in #test segments.

```
     ld  bcde,NNNN
     ld  bchl,NNNN
     ld  bcix,NNNN
     etc.

     ld  debc,(NN)
     ld  dehl,(NN)
     ld  deix,(NN)
     etc.

     ld  (NN),hlbc
     ld  (NN),hlde
     ld  (NN),ixiy
     etc.
```

```
push bcde
push dehl
push hlix
etc.

pop bcde
pop dehl
pop hlix
etc.
```

8080 assembler instructions

```
NOP

LXI R,D16   R = B D H SP
INX R
DAD R
DCX R

STAX B
STAX D
LDAX B
LDAX D

INR R       R = B C D E H L M A
DCR R
MVI R,D8

RLC
RRC
RAL
RAR
SHLD adr
DAA
LHLD adr
CMA
STA adr
STC
LDA adr
CMC

MOV R,R         R = B C D E H L M A
                except MOV M,M
HLT

ADD R           R = B C D E H L M A
ADC R
SUB R
SBB R
ANA R
XRA R
ORA R
CMP R

RET
RNZ
RZ
RNC
RC
RPO
RPE
RP
RM

JMP adr
JNZ adr
JZ adr
JNC adr
```

```
JC adr
JPO adr
JPE adr
JP adr
JM adr

CALL adr
CNZ adr
CZ adr
CNC adr
CC adr
CPO adr
CPE adr
CP adr
CM adr

RST N

POP R       R = B D H PSW
PUSH R

ADI D8
ACI D8
SUI D8
SBI D8
ANI D8
XRI D8
ORI D8
CPI D8

OUT D8
IN  D8
XTHL
PCHL
XCHG
DI
SPHL
EI
```

8080 assembler instructions for Z80 opcodes

This table lists the 8080 assembler syntax for the additional opcodes of the Z80 cpu.

They were rarely used, because people quickly switched over to the much more readable Zilog Z80 mnemonics. But there were some 8080 assemblers which added the new Z80 opcodes using 'their' syntax. It is absolutely not recommended to write new Z80 programs using 8080 assembler syntax, not even for writing new 8080 programs. Use Zilog Z80 syntax (the default for any Z80 assembler) instead.

Most mnemonics are taken from the CROSS manual except the following:

I doubt these were ever used…

```
RLCR r      CROSS-Doc: used RLC which is already used for RLCA, also deviation from
naming scheme
RRCR r      CROSS-Doc: used RRC which is already used for RRCA, also deviation from
naming scheme
OTDR        CROSS-Doc: used OUTDR which is a 5 letter word
OTIR        CROSS-Doc: used OUTIR which is a 5 letter word
DADX rr     CROSS-Doc: no opcode for ADD IX,rr
DADY rr     CROSS-Doc: no opcode for ADD IY,rr
PCIX        CROSS-Doc: no opcode for JP IX
PCIY        CROSS-Doc: no opcode for JP IY
INC  r      CROSS-Doc: no opcode for IN r,(c)
```

```
OUTC r      CROSS-Doc: no opcode for OUT (c),r
STAR        CROSS-Doc: no opcode for LD R,A
LDAI        CROSS-Doc: no opcode for LD A,I
LDAR        CROSS-Doc: no opcode for LD A,R
```

Some opcodes were extended to be used with dis(X) and dis(Y) as well.

The new registers I, R were not accessed by name but with dedicated mnemonics.

The index registers were abbreviated X and Y and an access (IX+dis) was written as dis(X).

New mnemonics

```
8080 syntax     Z80 syntax
DJNZ dis        djnz dis
JRZ  dis        jr   z,dis
JRNZ dis        jr   nz,dis
JRC  dis        jr   c,dis
JRNC dis        jr   nc,dis
JMPR dis        jr   dis
EXX             exx
EXAF            ex  af,af'
XTIX            ex  ix,(sp)
XTIY            ex  iy,(sp)
PCIX            jp  ix
PCIY            jp  iy
CCD             cpd
CCDR            cpdr
CCI             cpi
CCIR            cpir
LDI             ldi
LDIR            ldir
LDD             ldd
LDDR            lddr
IND             ind
INDR            indr
INI             ini
INIR            inir
OUTD            outd
OUTI            outi
OTDR            otdr            note: CROSS used OUTDR
OTIR            otir            note: CROSS used OUTIR
STAI            ld  i,a
STAR            ld  r,a
LDAI            ld  a,i
LDAR            ld  a,r
IM0             im  0
IM1             im  1
IM2             im  2
RETN            retn
RETI            reti
RLD             rld
RRD             rrd
NEG             neg
SPIX            ld  sp,ix
SPIY            ld  sp,iy
SBCD NN         ld  (NN),bc     named acc. to SHLD; note: *not* sbc!
SDED NN         ld  (NN),de     ""
SSPD NN         ld  (NN),sp     ""
SIXD NN         ld  (NN),ix     ""
SIYD NN         ld  (NN),iy     ""
LBCD NN         ld  bc,(NN)     named acc. to LHLD
LDED NN         ld  de,(NN)     ""
LSPD NN         ld  sp,(NN)     ""
LIXD NN         ld  ix,(NN)     ""
```

```
LIYD NN         ld  iy,(NN)     ""
INC  R          in  r,(c)       R = B C D E H L A;  note: *not* inc!
INP  R          in  r,(c)       ""
OUTC R          out (c),r       R = B C D E H L A
OUTP R          out (c),r       ""
DADC R          adc hl,rr       R = B D H SP
DSBC R          sbc hl,rr       R = B D H SP
DADX R          add ix,rr       R = B D X SP
DADY R          add iy,rr       R = B D Y SP
RES  N,R        res n,r         N = [0…7]; R = B C D E H L M A dis(X) dis(Y)

SET  N,R
set n,r         ""
BIT  N,R        bit n,r         ""
SLAR R          sla r           R = B C D E H L M A dis(X) dis(Y)
SRLR R          srl r           ""
SRAR R          sra r           ""
RALR R          rl  r           ""
RARR R          rr  r           ""
RRCR R          rrc r           ""
RLCR R          rlc r           ""
```

Existing mnemonics

Existing 8080 mnemonics which now also can be used with index registers

```
8080 syntax     Z80 syntax
ADD  R          add  a,r        R = B C D E H L M A dis(X) dis(Y)
ADC  R          adc  a,r        ""
SUB  R          sub  a,r        ""
SBB  R          sbc  a,r        ""
ANA  R          and  r          ""
ORA  R          or   r          ""
XRA  R          xor  r          ""
CMP  R          cmp  r          ""
INR  R          inc  r          ""
DCR  R          dec  r          ""
MVI  R,N        ld   r,N        ""
MOV  R,R        lr   r,r        ""; M, dis(X) or dis(Y) can only occur on one side
DCX  R          dec  rr         R = B D H SP X Y
INX  R          inc  rr         ""
LXI  R          ld   rr,NN      ""
PUSH R          push rr         R = B D H PSW X Y
POP  R          pop  rr         ""
```

Expressions

Numeric expressions

The assembler has an expression evaluator which supports various formats for numbers and operators with priority order. You may also use brackets, but don't start an expression for an immediate value with an opening bracket. The assembler thinks that an expression which starts with an opening bracket indicates direct memory addressing. If you really need to start an immediate value expression with brackets, precede it with the '+' monadic operator or a hash '#'.

Numeric literals

Numeric literals are the 'numbers' in an expression. They can be written in a multitude of formats:

```
12345    decimal number
12345d   decimal number
$1234    hexadecimal number
&1234    hexadecimal number  (since 4.3.4)
1234h    hexadecimal number
0x1234   hexadecimal number
%1010    binary number
1010b    binary number
0b1010   binary number
'a'      character literal
```

All number literals may be prefixed by a sign, either '+' or '-', which is actually handled as a monadic operator internally.

A character literal must be either a 7-bit ascii character or it is converted from utf-8. (note: all source files are expected to be either 7-bit ascii or utf-8 encoded.)

Character literals are translated to the target system's character set if the assembler directive '#charset' was used to define one. Else the character must be in range 0 to 255. If the character is not available in the target character set the assembler reports an error.

If the values are stored with 'defb' or similar, all values must fit in a byte each. Then characters like '€' will raise an error. If they are stored with 'defw' then '€' will be ok.

Labels

Of course labels can be used in expressions. Labels may refer to local or global symbols, depending on whether they were defined inside the current #local context (if any) or outside any local context. See chapter '#local' for more information.

Predefined labels

The assembler defines labels which can be tested with 'defined(NAME)' to test if certain command line options are set.

The handling of these predefined labels changed slightly in version 4.3.4.

_z80_ This label is defined if option --z80 is set on the command line or after the .z80 pseudo instruction. It is not set if target z80 is used by default.

_z180_ This label is defined if option --z180 is set on the command line or after the .z180 pseudo instruction.

_8080_ This label is defined if option --8080 is set on the command line or after the .8080 pseudo instruction. It is not set if target 8080 is used by default.

ixcbr2 This label is defined if the option '--ixcbr2' was set on the command line or after the pseudo instruction '.ixcbr2'.

ixcbxh This label is defined if the option '--ixcbxh' was set on the command line or after the pseudo instruction '.ixcbxh'.

Note, that source line 1 is also used to set command line options if it starts with a shebang "#!".

'$' and '.' (code address)

'$' and '.' are used to get the value of the current logical code position. It always refers to the start of a statement, e.g. to the first byte of an instruction or the postition of the first byte in a 'defb' instruction:

```
foo:    djnz $      ; jump to foo
bar:    defb $,$,$  ; stores 3x the value of bar
```

'.' is recognized for old sources only, do not use it in new code.

'$$' (physical code address)

'$$' refers to the physical code position. For an explanation of 'physical' vs. 'logical' code position see instruction '.phase'.

__line__

Inserts the current line number. E.g.:

```
.macro abort
    ld  de, __line__
    jp  abort
.endm
```

hi() and lo()

hi(NN) gets the high byte of a 2-byte word. Similar to NN >> 8.

lo(NN) gets the low byte of a 2-byte word. Similar to NN & 0xFF.

Function names must be lowercase except if --casefold or --asm8080 is selected.

E.g. if you have a table which must be located completely inside a page, then you can automatically raise the table base if the table would cross a page boundary: (note: use a calculated expression for .align if the code address is not yet valid in pass 1.)

```
.align hi($) != hi($+table_size-1)  ? 256 : 1
table:  ds  table_size
```

min() and max()

min(N1,N2) gets the value of the smaller of two values.

max(N1,N2) gets the value of the bigger of two values.

Function names must be lowercase except if --casefold or --asm8080 is selected.

You can also pass more than 2 arguments to min and max.

e.g. calculate size for a scratch pad:

```
scratch:   ds max(fp_scratch,txt_scratch,gfx_scratch)
```

sin() and cos()

sin(ANGLE,N2,N3) calculate the sinus of an angle.

cos(ANGLE,N2,N3) calculate the cosinus of an angle.

Function names must be lowercase except if --casefold or --asm8080 is selected.

The angle is scaled according to argument N2, which defines the value for a full circle. The result is scaled with argument N3. E.g. to generate a sine table with 256 values scaled to range ±127 use it in conjuction with pseudo instruction .rept:

```
angle = 0
.rept 256
    db  sin(angle,256,127)
angle = angle +1
.endm
```

opcode()

Function 'opcode(…)' can be used to get the major byte of an opcode. This is useful if you want to poke an opcode byte into some self-modifying portions of code or to skip over the first 1-byte opcode of a second entry point to a subroutine with db opcode(cp a,N), which is faster and shorter than jr $+3.

The 'major byte' is the first byte for most opcodes, the byte after 0xCB, 0xED, 0xDD or 0xFD for most others or the 4th byte of a 0xDD 0xCB or 0xFD 0xCB instruction. (note: 0xDD and 0xFD are the index register prefixes.)

'opcode' must be lowercase except if --casefold or --asm8080 is selected.

All opcodes for the Z80 cpu as well as the additional opcodes of the Z180 cpu are supported. If 8080 assembler is selected with --asm8080 then 8080 assembler mnemonics must be used.

Opcode names follow these rules:

- names should be in lower case
- arguments can be:
- a register name: af, af', bc, de, hl, sp, ix, iy, pc, a, f, b, c, d, e, h, l, r, i

for xh, xl, yh and yl simply use h and l.

- a flag name: z, nz, c, nc, po, pe, m, p
- register indirect: (bc), (de), (hl), (sp), (c)
- index register with offset: (ix+N), (iy+N) and variants: (ix+dis), (iy+dis), (ix+offs), (iy+offs)
- immediate value and memory: N, NN, (N), (NN) and variants: dis, offs
- fixed values: 0, 1, 2, 3, 4, 5, 6, 7, 8, 16, .. 56 and variants: $00, $08 .. $38, 0x00 … 0x38, 00h .. 38h

E.g.:

```
db opcode(nop)              just the name for opcodes without argument
db opcode(ld a,b)           name plus arguments which go into the opcode
db opcode(bit 3,(hl))       '3' is part of the opcode!
db opcode(rst 8)
db opcode(ld a,N)
db opcode(ld a,(NN))
db opcode(ld a,(bc))
db opcode(ld a,(ix+offs))   you can use 'ld a,(hl)' here as well
db opcode(jr c,NN)
db opcode(ex af,af')
```

Examples:

```
; jump over an opcode:
    db  opcode(cp a,N)  ; shorter and faster than 'jr'
1$: ld  a,e

; self modifying code:
    ld  (opcode1), opcode(inc l)
    ld  (opcode2), opcode(inc h)
```

defined(LABEL)

This results in 0 or 1 depending on whether the named label is already defined at this position in source.

'defined' must be lowercase except if --casefold or --asm8080 is selected.

note: use of this function in a local context may malfunction if a local name of this name is defined after this test.

E.g.:

```
#if defined(logline)
    ld  hl, __line__
    call    logline
#endif
```

required(LABEL)

This results in 0 or 1 depending on whether the named label was used but not yet defined at this position in source.

E.g.:

```
#if required(mul_hl_de)
```

```
     #include "lib/mul_hl_de.s"
     #endif
```

target(NAME)

Test whether the named #target is selected. Use this if you made it selectable for some reason in your source. Result is 0 or 1.

Possible targets are: ROM, BIN, Z80, SNA, TAP, TAPE, O, P, 80, 81, P81 and ACE.

'target' must be lowercase except if --casefold or --asm8080 is selected.

Example:

```
     ; include a tape header if #target is set to TAPE:
     #if target(TAPE)
     #code CODE_HEADER, 0, 17, headerflag
         defb    3, "mcode     "
         defw    code_size,load_address,0
     #endif
```

segment(NAME)

Test whether the named segment is the currently selected #code or #data segment. Result is 0 or 1.

'segment' must be lowercase except if --casefold or --asm8080 is selected.

Example:

```
     ; don't include 'ret' opcode if this is in boot code:
     #if !segment(GSINIT)
         ret
     #endif
```

Operators

The following operators are recognized and listed in order of precedence:

```
     brackets            ( ... )
     monadic operators   + - ~ !        plus sign, minus sign, 2's complement, negation
     shifting            << >>          shift left or right; left operand is target
                         shl shr
     bit masking         & | ^          bitwise and, or, xor
                         and or xor
     mult/div            * / %          multiply, divide, remainder ('\' is no longer
                                        recognized)
     add/sub             + -            add, subtract
     comparisions        > < >= <= = <> greater than etc.; result is 0 or 1
                         == != eq ne
                         gt lt ge le
     boolean             && ||          pruning
     selector            ? :            pruning
```

Since 4.0.21: The backslash '\' is no longer recognized as the remainder operation. The backslash is now used to separate multiple opcodes in one line.

Up to 4.0.22: The first argument of a pruning operator must be valid in pass 1.

Since 4.0.23: The first argument of a pruning operator should be valid in pass 1.

If it isn't, then both sub expressions are evaluated in pass 1. If there is an undefined label referenced in a later pruned path it will be still required and the assembler will report it as undefined.

If you used command line option '--flatops' then all operators are evaluated strictly from left to right.

String expressions

```
defm "<some text>" [ + <value> ]
defm '<some text>' [ + <value> ]
```

Strings cannot be stored in labels and therefore most string operations make no sense. Strings can just occur as a string literal, either enclosed in a pair of "' or a pair of '"'. Whichever character is chosen it must not occur inside the string.

Note: special characters in string cannot be escaped with '\'. This syntax was (probably) invented later for the programming language 'C'.

```
defm "say 'Hello!'",0
defm 'say "Hello!"',0
```

Characters in the string are translated to the target system's character set if the assembler directive '#charset' was used to define one. Else the characters must be in range 0 to 255. If the character is not available in the target character set the assembler reports an error.

The last character of a string may be modified by a numeric operation, most commonly used is setting bit 7 of the last character:

```
defm "foobar"+0x80
```

The following operators are allowed: '+', '-', '&', '|', '^'.

There are 4 predefined strings available in the 'defm' instruction:

__date__, __time__, __file__ and __line__.

Including C Source Files

This is a summary page for all the c source related information otherwise spread over the zasm documentation.

Related command line options

```
-c path/to/cc       set path to c compiler (default: sdcc in $PATH)
-t path/to/dir      set path to temp dir for c compiler (default: output dir)
-I path/to/dir      set path to c system header dir (default: sdcc default)
-L path/to/dir      set path to standard library dir (default: none)
```

Related assembler directives

```
#include
#include library
#cpath
#cflags
#target, #code and #data
#local and #endlocal
```

Example sources:

```
Examples/template_rom_with_c_code.asm
Test/test-tap.asm
```

include/ and lib/ directory

Note: the supplied sdcc/lib/ directory requires sdcc 3.4.1.

```
sdcc/include/
sdcc/lib/
```

Interesting reading

```
sdcc_info.txt
sdcc_manual.pdf
```

Command line options for C compiler

-c /path/to/cc

If you include c sources and if the c compiler cannot be found in your $PATH or if you want to use a different one, then you can tell the assembler which executable to use for compiling c sources.

If a partial path is used then your current working directory applies.

-t /path/to/dir

Define the directory to use for temporary files. The only temp files created by zasm are those generated by the c compiler which will be stored in subdirectory 's/' inside this filder.

The default location is the output file's directory.

If a partial path is used then your current working directory applies.

-I /path/to/dir

Define the path to system header files for the c compiler. Normally the c compiler already knows where they are, but if you want to use a different directory or if your headers just are not in the standard location, you can use this option. It will pass two arguments to the c compiler:

```
/bin/sdcc --nostdinc -I/path/to/dir
```

The first one tells it to forget about the standard location for header files and the second one tells it where they actually are.

If a partial path is used then your current working directory applies.

Note: include directories can also be added with the assembler directive #CFLAGS in your source file. There a partial path refers to the source file's directory!

Note: **mind the space!**

-L path/to/dir

Define the path to the system's library directory. This directory is used by zasm to resolve missing symbols in assembler directive '#include system library'.

The default location is derived from the include directory path. If the include directory path was not specified on the command line then the library path must be specified or '#include system library' won't work. But you still can hard code the path in the #include statement.

If a partial path is used then your current working directory applies.

Note: **mind the space!**

Short tutorial for using C files

zasm can inclue C source files in your project. These will be compiled using the C compiler sdcc.

In a 'normal' C project you typically start writing C sources and then at some point wonder what the file crt0.s may do and then how to modify it. With zasm you start with an assembler file which mostly does what the crt0.s file should do. zasm also resolves global symbols from the system libraries and links your project into one binary.

The main assembler file must define a #target, some #code and #data segments, implement the restart vectors, an interrupt and the nmi handler, initialize the system hardware, initialize variables and define _putchar and _getchar, if required.

Find the C compiler

When it comes to compiling a C source, zasm looks for the C compiler. If it is not in your $PATH then you must tell zasm where it is with the command line option -c.

Example:

```
$> zasm -c /usr/bin/sdcc foo.asm
```

Locate the system headers

Next the C compiler needs to know where the system headers are when a C source includes some of them. The C compiler will look at the default locations, but you may pass it the include/ directory from the zasm distribution, which can be done with command line option -I.

Example:

```
$> zasm -I /work/sdcc/include foo.asm
```

Required segments

The C compiler sdcc expects some segments to be defined in your source.

Example:

```
_ram_start equu 0x4000   ; wherever
#target rom             ; define the code model
#code _HOME,0           ; code that must not be put in a bank switched part of
memory.
#code _GSINIT           ; init code: the compiler adds some code here and there as
required
#code _CODE             ; most code and const data go here
#code _INITIALIZER      ; initializer data (in rom) for initialized variables (in
ram)
#code _CABS,*,0         ; referenced but never (?) actually used by sdcc
#code _GSFINAL,*,0      ; referenced but never (?) actually used by sdcc
#data _DATA, _ram_start ; uninitialized variables
#data _INITIALIZED      ; variables initialized from _INITIALIZER
#data _DABS,*,0         ; referenced but never (?) actually used by sdcc
#data _RSEG,*,0         ; referenced but never (?) actually used by kcc
```

In this example the _HOME segment starts at address 0 and therefore must start with the handlers for the restart vectors. As an example see Examples/template_rom_with_c_code.asm.

Hint: as one of the very last lines in your source add a ret statement or directly a call to main() in _GSINIT:

```
#code _GSINIT           ; switch to segment _GSINIT
    call _main          ; final action in _GSINIT: call main()
    rst  0              ; if it returns (which it shouldn't)
```

How to include C source files

The major file of your project must be an assembler file.

This file can include other assembler files or c source files.

C source files must end with the file name extension '.c'.

Example:

```
#include "example.c"
```

How to resolve symbols from the system library

Compiled C source files frequently contain references to procedures or other global symbols in the system libraries. These can be resolved automatically if your assembler file includes the system library with the #include assembler directive after the last included c source file or after the last reference to an undefined symbol from the system library for the first time.

Example:

```
#include standard library
```

This will resolve all symbols from the system library known as required at that point in your source.

As a prerequisite zasm must know where to find the system library directory. It is recommended to use the supplied directory from the distribution and then you have to tell zasm where it is. This can be done with the command line option -L.

Example:

```
$> zasm -L /work/sdcc/lib foo.asm
```

Alternatively you can give a path in the '#include library' directive itself:

```
#include library "/work/sdcc/lib"
```

Using a heap

If you use malloc() etc. then you also need a heap. The actual requirements depend on the implementation of malloc() and free() and this is how it currently should work: (else refer to the implementations of lib/_malloc and lib/_mfree…)

Define data segment _HEAP as the last data segment if you want it to occupy all unused ram up to ram end.

```
#data _HEAP                 ; heap:
__sdcc_heap_start:          ; --> sdcc _malloc.c

ds  _min_heap_size      ; minimum required size
ds  _ram_end-$-1        ; add all unused memory to the heap
__sdcc_heap_end:            ; --> sdcc _malloc.c
ds  1
```

Initialization code

System initialization must be done in assembler as far as it cannot be done in C. An important step in system initialization is the initialization of initialized variables. After that main() must be called which should not return.

Example:

```
_init:                      ; jump here from reset entry at 0x0000
    di
    ld  sp,_ram_end         ; set stack pointer

    ld  bc,_initializer_len ; length of segment _INITIALIZER (must be calculated)
    ld  de,_INITIALIZED     ; start of segment _INITIALIZED
    ld  hl,_INITIALIZER     ; start of segment _INITIALIZER
    ld  a,b
    or  c
    jr  z,$+4
    ldir                    ; copy initializer data into initialized variables

    call _GSINIT            ; Initialize global variables (whatever the c compiler
                            ; has put in here)
    call _main              ; call main() (if not done at the end of _GSINIT
                            ; itself)
    rst  0                  ;
if it returns
```

How to implement _putchar() and _getchar()

C will call the symbols _putchar and _getchar if your source includes any print or read function call. These cannot be provided by the standard system library because they are highly specific to your target system. If C sources link against these symbols then you must provide appropriate definitions.

Eventually you can implement _putchar and _getchar in C, but if you need to implement them in assembler, then you need to know how the C compiler passes the arguments.

Argument passing to and from functions may vary with compiler options, so i recommend that you first include dummy definitions for _putchar() and _getchar() in your project and inspect the list file generated by the assembler to see what the compiler does.

Example: in a C file define:

```
    char inchar,outchar;
    char getchar(void) { return inchar; }
    void putchar(char c) { outchar = c; }
```

assemble your project and you may see the following assember source for _putchar and _getchar:

```
    ;char getchar(void) { return inchar; }
    _getchar:
        ld  hl,#_inchar
        ld  l,(hl)
        ret

    ;void putchar(char c) { outchar = c; }
    _putchar:
        push ix
        ld   ix,#0
        add  ix,sp
        ld   a, 4 (ix)
        ld   (#_outchar),a
        pop  ix
        ret
```

Use this as a template for your own implementation. For an example see Examples/zx_spectrum_io_rom.s

8080 Assembler

```
$> zasm --asm8080 mysource.asm
```

zasm can assemble source which is in the historic 8080 assembler format if it is started with command line option '--asm8080'.

zasm supports most of a typical 8080 assembler source files, but only as an addition to it's standard Z80 source handling. It expects one instruction per line, comments start with a semicolon ";" and label definitions should start in column 1 except if you also define '--reqcolon'.

All identifiers (mnenonics, register names and directives) and label names are not case sensitive!

The 8080 assembler is provided to assemble existing source. New programs should be written in the much better readable Z80 syntax. zasm can assemble Z80 source files restricted to the 8080 instruction set and registers if started with command line option '--8080'.

8080 code example

```
inpl2:  LXI     H,IBUFF         ;Input buffer addr
        SHLD    IBUFP
        MVI     C,0             ;Init count to zero

inpli:  CALL    intt            ;Get char from console

        CPI     ' '             ;Control char?
        JC      inplc           ;Yes

        CPI     DEL             ;Delete char?
        JZ      inplb           ;Yes

        CPI     'Z'+1           ;Upper case?
        JC      inpl3           ;Yes
        ANI     5Fh             ;No - so make upper case

inpl3:  MOV     M,A             ;Into buffer
        MVI     A,IBL           ;Buffer size
        CMP     C               ;Full?
        JZ      inpli           ;Yes, loop
        MOV     A,M             ;Get char from buffer
        INX     H               ;Incr pointer
        INR     C               ; and count
inple:  CALL    OUTT            ;Show char
        JMP     inpli           ;Next char
```

8080 pseudo instructions

8080 pseudo instructions occure at the position of a real 8080 opcode and must normally be preceded with some spaces, except if command line option '--reqcolon' is used.

zasm knows the following 8080 assembler pseudo instructions:

ORG <nnnn>

Set the logical origin (code address) for the following code.

```
ORG 0C000h
```

END

Define the logical end of your source. END is optional.

IF .. ENDIF

Conditionally include a range of source source lines.

IF and ENDIF are just an alias to #IF and #ENDIF and therefore may be used interchangeable with #IF, #ELIF, #ELSE and #ENDIF. This may change.

```
IF option_foo=1
   IF option_bar
       ; <-- some code -->
   ENDIF
endif
```

DB

Insert bytes. DB is handled like 'defb' or 'defm' in Z80 assembler: You can put in strings here which may be enclosed in ' or ".

```
DB  lf,cr,'Hello You: ',0
```

DW

Insert words. Like 'defw' in Z80 assembler.

```
DW  foo, bar*2, stuff+33, 0xFFFF
```

DS

Insert space. Like 'defs' in Z80 assembler.

```
DS  66h - $
```

MACRO

Define a macro which can be invocated later like a 8080 instruction. The macro definition is terminated by the ENDM instruction. Macros may have arguments.

```
foo     MACRO   ARG1, ARG2
        mov a,&ARG1
        jmp &ARG2
        ENDM
        ...
        foo 42,$0008        ; use macro
```

REPT

Define a range of source lines which shall be repeated multiple time.

```
REPT $2000 - $
DB   0
```

```
     ENDM
```

Labels

EQU

Assign a value to a label. Behaves like 'equ' for the Z80 assembler.

```
     foobar  equ 5
       foo   equ 2
       bar   equ foobar - FOO
```

In contrast to the Z80 assembler, label names are not case sensitive!

Register names can be used as label names. (but, really really, shouldn't.) The names for the 8080 assembler directives are not allowed for label names.

Program labels must start in column 1 and may be followed by a colon ':' (optional), except if option '--reqcolon' was used. Then they may start in any column and the colon is required. (sic!)

```
     start:  CALL  fii
             CPI   'Z'
             JC    in3
             ...
     in3:    ...
```

SET

Define a redefinable label. Most useful if used with macros

```
     foo     set 1
             ...
     foo     set foo+1
```

Expressions

Basically all possibilities of the Z80 assembler are allowed, because the expression parser of the Z80 assembler is used.

Convert 8080 to Z80 assembler syntax

Since version 4.3.0: If zasm is started with command line option '--convert8080', it will convert the supplied source file to Z80 syntax, which is much better readable. zasm converts the source and writes it to the output file or a file with a derived name. Then it assembles the original source and the converted source and compares the outputs. The Z80 source is assembled with option '--casefold'. Flags required to assemble the 8080 source – e.g. '--reqcolon' – must also be used for the conversion. The source is converted independently of any successive errors.

Output file: if no output is specified, the output file is written to the same folder as the input file. If no output filename is given, then zasm derives the filename for the Z80 source by appending "_z80" to the file's basename.

After creating the Z80 source file, zasm assembles both files. The binary file goes to the temp folder which is, by default, the same as the output folder. Only one binary file for the 8080 source is created in this directory, the second binary is compared to this file directly. (actually, a temp file is created in /tmp/zasm/.) If list files are enabled (the default), then list files are created for both assemblies. You can disable listfiles with option '-l0'.

Only source files for the i8080 can be converted. The 8080 mnemonics for the additional Z80 opcodes are not supported.

The generated source should be assembled with command line option '--casefold', because label names for 8080 assemblers were typically case-insensitive. Additionally, any other options like '--reqcolon', which were required for the 8080 source, are also required for the Z80 source.

If the source starts with a shebang, zasm also modifies assembler options found in the first line. Specifically, it removes '--asm8080', '--convert8080' and adds '—casefold'.

Targets

Overview

The target for this source is set by the #target assembler directive. Available targets are:

```
bin rom sna z80 tap ace o 80 p 81 p81
```

The target also defines the default file name extension for the output file. Targets bin and rom can also be written in Intel Hex or Motorola S-Record format, if the command line options '-x' or '-s' are given.

If a target is defined, then at least one #code segment must be defined as well. Depending on the target more may be required and certain additional requirements may apply.

Instead of using the #target/#code metapher, simple projects may omit them and just define an origin before the first actual code.

Example:

```
#target rom
#code _EPROM,0,0x2000
reset:  di
        jp  _init
        ...
```

General rules

The target file format must be set with the #target assembler directive.

```
#target <extension>
```

This does not define whether the generated data should be written as a plain binary file or in one of the hex file formats. This must be selected by the '-x' or '-s' command line option. The assembler defaults to a binary file.

```
#code <name>, <start>, <size> [,<flag>]
```

This directive defines an object code segment and set's the physical and the initial logical code address to <start> and defines a maximum segment size <size>.

<start> does not define the position of the code in the output file. It should define the address where this segment is visible to the cpu.

Multiple code segments are possible and are simply concatenated. Code segments are either padded at the end to the declared maximum size or they are truncated after the last byte stored, if the size is omitted.

#target RAM (old: BIN) and ROM

These are the simplest and most common targets. ROM is intended for creating rom files while RAM is intended to be downloaded somehow into the ram of a target system. One difference between ROM and RAM is the default fill byte used to fill gaps, e.g. in the 'defs' pseudo instruction or for the padding at the and of a #code segment. The main difference (since version 4.2.8) are the addresses stored in hex files. Target RAM stores addresses based on the origin of each segment while ROM stores consecutive addresses starting at $0000 suitable for an eprom burner.

#target ROM

This is also the default target if no target is set, e.g. in legacy source.

This file is a plain rom or eprom image intended to be used with an eprom burner.

The default fill byte for .defs and for padding segments is $FF.

Multiple code segments are simply concatenated to create the binary output file.

Hex files are also created by simply concatenating all code segments.

The padding at the end of each segment (if any) is not stored.

The addresses in the hex files start at $0000 and define the code storage position in the eprom or memory file, suitable for use by an eprom burner.

#target ROM files can include multiple segments mapping to the same cpu address. (paging)

```
#target rom
#code   <name>,<start>,<size>
...
```

Example for a rom which is paged into the Z80 address space in 2 pages at address $C000:

```
#target rom
#code   PAGE1, $C000, $4000
        ...
#code   PAGE2, $C000, $4000
        ...
```

#target RAM

until version 4.2.7:

Target RAM behaves identical to target ROM except for the default fill byte which is is $00.

since version 4.2.8:

This file is a plain memory dump intended to be loaded by a ram loader on the target system.

The default fill byte for .defs and for padding segments is $00.

Binary files for ROM and RAM are identical except for the default fill pattern:

multiple code segments are simply concatenated.

Hex files are created by simply concatenating all code segments.

The padding at the end of each segment (if any) is not stored.

Addresses in hex files are based on the origins of the #code segments, making it suitable for a ram loader.

Caveat: Multiple segments are loaded differently for binary and hex files!

A binary file is just a concatenation of all segments in order of definition in the source file, disregarding the segment start addresses. Eventually the segments are copied to their final position by the bootstrap code.

A hex file will most likely be directly loaded to the final position by a ram loader using the addresses in the hex file, even if segments are defined in arbitrary order or with unused space between them.

```
#target ram
#code   <name>,<start>,<size>
        ...
```

Writing Intel Hex files

Using command line option '-x' the RAM and ROM targets can be written in the Intel Hex format. These files look like this:

```
:200000003E00ED79ED79ED793E40ED793EDAED793E37ED79ED780F30FB3A1027D370ED787B
:090020000F0F30FADB70321027DB
:00000001FF
```

Trailing fill bytes (0xFF for ROM and 0x00 for RAM) in each segment are not stored in the file. This generally reduces file size and the time to burn the file into an eprom or to transmit it to the host system. Therefore the target system should erase the ram to 0x00 before downloading a hex file. Eproms must be erased before they can be burned and they are erased to 0xFF. That's why the fill byte for ROM is 0xFF.

Writing Motorola S-Record format

Using command line option '-s' the RAM and ROM targets can be written in the Motorola S-Record format. These files look like this:

```
S00F000073203230031352D30312D313178
S12C00003E00ED79ED79ED793E40ED793EDAED793E37ED79ED780F30FB3A1027D370ED780F0F30FADB7
032102772
S5030001FB
S9030000FC
```

Trailing fill bytes are handled as for the Intel hex format.

#target SNA

A ZX Spectrum NMI snapshot file.

The .sna snapshot file describes a ZX Spectrum 48k serving an NMI request. There are also variants for 16k or 128k Speccies, but zasm currently only supports the basic 48k variant. (send email if you need the other variants!)

Information on the ZX Spectrum file formats can be found in the ZX Spectrum FAQ which can be found at WOS.

This file consists of a header section which mostly stores the Z80 registers and a ram dump from location $4000 to $FFFF. This is reproduced in the assembler source file as 2 code sections:

```
#target sna
#code   _HEADER,0,27
    ...
#code   _DATA,0x4000,0xC000
    ...
```

See the zxsp .sna template file: template_sna.asm

The first code segment must be exactly 27 bytes long. zasm will validate some of the bytes stored here.

The start address of the ram segment must be $4000 and the length must be 0xC000 bytes. The ram segment may be split into multiple segments, but they must start at 0x4000 and sum up to a total of 0xC000 bytes. zasm will check this.

The ram contents are written to the file in the uncompressed .sna format. The instruction pointer PC is not stored in the header segment but on the machine stack in ram and will be popped from it as a first action after loading this file into the emulator.

#target Z80

A ZX Spectrum emulator snapshot.

Information on the ZX Spectrum file formats can be found in the ZX Spectrum FAQ which can be found at WOS.

A .z80 file may contain a snapshot for most ZX Spectrum models and, on some emulators, e.g. zxsp for OSX, for a bunch of clones, ZX80 and ZX81 variants and the Jupiter Ace as well.

The .z80 file starts with a header section which is followed by multiple ram chunks. This is reproduced as one header segment and an appropriate number of appropriately sized segments with ram contents.

The basic layout is like this:

```
#target z80
#code   _HEADER,0,size      ; size depends on version
    ...

#code   _RAM,0x4000,0xC000,<id>
    ...
```

See the zxsp .z80 template file: template_z80.asm

A version 1.45 snapshot

The header size is 30 bytes and the ram chunk must be a single segment of 0xC000 bytes. Use of version 1.45 is discouraged.

Note: unlike the other versions the version 1.45 ram segment has no ID flag.

```
#target z80
#code   _HEADER,0,30
```

```
        ...
    #code   _RAM,0x4000,0xC000
        ...
```

A version 2.0.1 snapshot

The header section contains additional information and the ram is saved in pages of 0x4000 bytes each. This has to be reproduced in the source file. Depending on the model declared in the header section, (see template_z80.asm) an appropriate number of correctly sized code segments must follow which must have an additional flag argument which defines the ID which is assigned to their ram chunk.

This would be appropriate for a ZX Spectrum 48k:

```
    #target z80
    #code   _HEADER,0,55
        ...
    #code   _RAM1,0x4000,0x4000,8
        ...
    #code   _RAM2,0x8000,0x4000,4
        ...
    #code   _RAM3,0xC000,0x4000,5
        ...
```

evtl. i will implement the 'varying blocksize' feature from the b&w models for zxsp and allow zasm to produce such files. this would allow one continuous segment to be used.

A version 3.0 file is very similar, except that some few model IDs have changed and more data is added to the header section.

This would be appropriate for a ZX Spectrum 128k:

```
    #target z80
    #code   _HEADER,0,86
        ...
    #code   _RAM0,0xC000,0x4000,3   // ram mapped at 0xC000 after reset
        ...
    #code   _RAM1,0xC000,0x4000,4
        ...
    #code   _RAM2,0x8000,0x4000,5   // ram at 0x8000
        ...
    #code   _RAM3,0xC000,0x4000,6
        ...
    #code   _RAM4,0xC000,0x4000,7
        ...
    #code   _RAM5,0x4000,0x4000,8   // ram at 0x4000
        ...
    #code   _RAM6,0xC000,0x4000,9
        ...
    #code   _RAM7,0xC000,0x4000,10
        ...
```

black&white models

The b&w models have varying ram sizes from 1k to 64k or more. To allow storing of any arbitrary ram size the memory is chunked into blocks of 1k, 2k, 4k, 8k, 16, 32k and 64k. Currently each block must occur at most once. So 1k, 2k or 16k ram would be saved as 1 block while 48k would be saved as 2. The block IDs can be picked from template_z80.asm.

```
    #code   _HEADER 0,55
```

```
        ...
    #code   _RAM,0x4000,0x0800,4    ; looks like a TS1000 ;-)
        ...
```

#target TAP or TAPE

```
    ; ZX Spectrum tape file:
    #target tap
    #code <name>, <start>,<len>,<flag>
    #code <name>, <start>,<len>,flag=<flag>
    #code <name>, <start>,<len>,FLAG=<flag>
    ...
    ; Jupiter Ace tape file:
    #target tap
    #code <name>, <start>,<len>,flag=none
    #code <name>, <start>,<len>,none
```

This target defines a ZX Spectrum or a Jupiter Ace tape file. This file may and should contain several code segments. Normally each code segment defines one block on the tape. The blocks are written to the file in the uncompressed TAP format.

The tape file format represents programs and data saved to a music compact cassette by the original ZX Spectrum tape saving routines.

ZX Spectrum vs. Jupiter Ace tap format

Unluckily the tape file formats for the ZX Spectrum and for the Jupiter Ace are only similar, not identical. zasm handles this difference for you, but it has to use a little heuristics to decide in which format to save.

Normally a ZX Spectrum file is written. Except

if the following conditions are met, then a Jupiter Ace tape file is written:

Since version 4.2.0 zasm uses a different method to select Jupiter Ace .tap format:

- If you set the flag byte in the #code directive to none, then a Jupiter Ace-style tape block is written. This block does not store the flag byte (therefore 'none') but instead emulators will implicitely assume $00 and $FF for the flag byte in alternating order.

Layout of a block in a ZX Spectrum tape file:

```
    defw    <length>
    defb    <flag>
    defm    <data>
    defb    <checksum>
```

Layout of a block in a Jupiter Ace tape file:

```
    defw    <length>
    defm    <data>
    defb    <checksum>
```

The flag argument defines the type of block and is set from the #code segment's flag byte.

Then the raw data follows which is taken from the #code segment's data.

Finally a checksum follows which is calculated and appended by the assembler.

A tape file is a simple sequence of any number of these blocks.

Any kind of data is typically saved in two blocks: a header block, containing information about the following data, and a data block, containing data as described by the preceding header.

A complete game for the ZX Spectrum is typically saved in two parts: a basic loader, which consists of a header and a data block and the machine code part, which consists of a header and data block too. The basic part is typically saved with an auto start address which loads the following parts and starts the game.

A game for the Jupiter Ace may consist of only a single part (which consists of a header and data block). Machine code was not such a requirement because Forth is already pretty fast. The Jupiter Ace did not support auto starting and therefore the user had to type commands to load following blocks (if any) and to start and even restart a game.

The flag argument defines the type of block. On both the ZX Spectrum and the Jupiter Ace this is typically $00 for a header block and $FF for the following data block.

For an example see the .tap template file: template_tap.asm

It is possible to spread a tape block across multiple #code segments. This is required if you want to include c source files. ~~The first #code segment must define an address and the block flag and the following #code segments must have exactly matching addresses (you may put in '*' for the address) and no flag. All #code segments which meet this requirement will be united with the first segment to one tape block.~~

New since version 4.0.24: Each code segment which has a flag byte starts a new tape block. All following segments which have no flag byte are appended to this tape block. A segment address defined for a following block does not create a 'gap' on the tape. The segment will be loaded where it is loaded and the program is assumed to move it to the defined location befor it is used. Actually this is true for the first block too: The tape loader defines where the block will be loaded, not the segment address stated in the #code directive. This can be used if you want to move the entire program to a lower location in ram when it starts up.

Basic layout of a ZX Spectrum tape

```
#target tap
;
headerflag:     equ 0
dataflag:       equ 0xff
;
; use printer buffer for variables:
;

#data VARIABLES, 0x5B00, 0x100
;
; Basic loader, header:
;

#code PROG_HEADER,0,17,headerflag
        defb    0                       ; Indicates a Basic program
        defb    "mloader   "            ; the block name, 10 bytes long
        defw    variables_end-0         ; length of block = length of basic program
                                          plus variables
        defw    10                      ; line number for auto-start, 0x8000
```

```
                                          if none
        defw    program_end-0           ; length of the basic program without
                                          variables
;
; Tokenized Basic program:
;

#code PROG_DATA,0,*,dataflag
;
; 10 CLEAR 23999
        defb    0,10                    ; line number
        defb    end10-($+1)             ; line length
        defb    0                       ; statement number
        defb    tCLEAR                  ; token CLEAR
        defm    "23999",$0e0000bf5d00   ; number 23999, ascii & internal format
end10:  defb    $0d                     ; line end marker
;
; 20 LOAD "" CODE 24000
        defb    0,20                    ; line number
        defb    end20-($+1)             ; line length
        defb    0                       ; statement number
        defb    tLOAD,'"','"',tCODE     ; token LOAD, 2 quotes, token CODE
        defm    "24000",$0e0000c05d00   ; number 24000, ascii & internal format
end20:  defb    $0d                     ; line end marker
;
; 30 RANDOMIZE USR 24000
        defb    0,30                    ; line number
        defb    end30-($+1)             ; line length
        defb    0                       ; statement number
        defb    tRANDOMIZE,tUSR         ; token RANDOMIZE, token USR
        defm    "24000",$0e0000c05d00   ; number 24000, ascii & internal format
end30:  defb    $0d                     ; line end marker
;
program_end:
;
; <-- Basic variables -->
;
variables_end:

;
; Machine code block, header:
;

#code CODE_HEADER,0,17,headerflag
        defb    3                       ; Indicates binary data
        defb    "mcode     "            ; the block name, 10 bytes long
        defw    code_end-code_start     ; length of data block which follows
        defw    code_start              ; default location for the data
        defw    0                       ; unused
;
; Machine code block, data:
;

#code CODE_DATA, code_start,*,dataflag
;
; <-- Z80 assembler code and data -->
;
code_end:
```

#target TZX

```
#target tzx
#code <name>, <start>,<len>,<flag>
#code <name>, <start>,<len>,<flag>
#code <name>, <start>,<len>,<flag>,...
...
```

Introduced in version 4.2.0

#target TZX creates a tape file for the ZX Spectrum, ZX81, Amstrad CPC, SAM Coupé, Jupiter ACE and Enterprise or similar computers. This tape file format represents programs and data saved to a music compact cassette by the original ZX Spectrum or custom tape saving routines.

While a TAP file can only contain standard data blocks as saved with the original ZX Spectrum "save tape" routine, TZX files can contain data blocks with custom encoding, e.g. turbo loading blocks or ZX81-like blocks, additional emulator information and custom audio sequences. TZX files were originally used for preserving and archiving real ZX Spectrum tapes.

Typical layout of a game tape

The assembler source file will probably contain several #code segments. Normally each #code segment defines one block on the tape.

- • A normal block consists of a pilot tone (lead-in), two sync pulses, a flag byte, the raw data bytes and a final checksum byte.
- • The flag argument defines the type of the block and is set in the #code directive.
- • Then the raw data follows which is the assembled data of this #code segment.
- • Finally a checksum is appended which is calculated by the assembler.

A tape file is a simple sequence of any number of these blocks.

Any kind of data was saved by the ZX Spectrum rom routine in two blocks: a header block, containing information about the following data, and a data block, containing data as described by the preceding header.

A complete game for the ZX Spectrum is typically saved in two parts: a Basic loader, which consists of a header and a data block and the machine code part, which consists of a header and data block too. The Basic part is typically saved with an auto start address which loads the following parts and starts the game.

The flag argument defines the type of block. This is typically $00 for a header block and $FF for the following data block.

zasm allows you to spread a tape block across multiple #code segments. This is useful if you want to include C source files. Each #code segment with a flag byte starts a new tape block. All following #code segments with no flag byte are joined with this tape block. A segment address defined for a following block does not create a 'gap' on the tape. The segment will be loaded where it is loaded and the program is assumed to move it to the defined location befor it is used. Actually this is true for the first block too: The tape loader defines where the block will be loaded, not the segment address stated in the #code directive. This can be used if you want to move the entire program to a lower location in ram when it starts up.

For an example of the basic layout of a ZX Spectrum tape see #target TAP.

Supported TZX block types

TZX files are made out of blocks. Each of them may, or may not, define a complete data block on the tape. There are various block types which are identified by a block ID:

Supported block types

```
     0x10: Standard speed data block
     0x11: Turbo speed data block
     0x12: Pure tone
     0x13: Sequence of pulses of various lengths
     0x14: Pure data block
     0x18: CSW recording block
     0x19: Generalized data block
     0x20: Pause (silence) or '
     Stop the tape' command
     0x21: Group start
     0x22: Group end
     0x24: Loop start
     0x25: Loop end
     0x2A: Stop the tape
     if in 48K mode
     0x2B: Set signal level
     0x30: Text description
     0x31: Message block
     0x32: Archive info
     0x33: Hardware type
```

Using standard encoding scheme and timing

A TZX file represents what is audible on the tape, not neccessarily a specific meaning for the pulses. Therefore it is possible that the first byte recorded on tape is not a flag byte and the last byte is not the standard checksum and even some odd bits may appear at the end of a block which do not sum up to a full byte. You probably won't need these fancy options, therefore they are all optional.

#target TZX introduces a new syntax for optional arguments which is shown below in the second version of the #code directive. The square brackets indicate optionality. Round brackets with a vertical bar somewhere in the middle indicate a choice. Keywords are written in uppercase, but are also recognized in lowercase by the assembler.

```
     #CODE name, address, length, flag
     #CODE name, address, length, FLAG=(flag|NONE), [PILOT=(NONE|count)],
                          [CHECKSUM=NONE|ACE], [LASTBITS=lastbits], [PAUSE=pause]
```

FLAG: The value for the flag byte is in range 0 to 255. If the keyword NONE is used then zasm does not store a flag byte at the start of the tape block.

PILOT: If set to NONE then no pilot tone and no sync pulses are stored. If a number is given, then a pilot tone with that number of pulses is stored. The default is to store a data block which includes a pilot tone with 8063 or 3223 pulses depending on the flag byte.

CHECKSUM: If set to NONE then zasm will append no checksum after the data. The default is to append the standard ZX Spectrum checksum.

LASTBITS: The number of bits actually played from the last byte. Possible values are from 1 to 8. The upper bits of the last byte are used, because the bits were sent from msb to lsb by the ZX

Spectrum tape routine. Use of this option is incompatible with a checksum. If LASTBITS is omitted, then the last byte is stored (and played) as usual.

PAUSE: Length of the silent gap after this block in ms. If PAUSE is omitted then the default is 1000ms after a header block and 2000ms after a data block.

If CHECKSUM=ACE is defined, then some default values are changed:

- The checksum is calculated for the Jupiter Ace.
- Number of pilot pulses: 8192 (header) or 1024 (data).
- Length of pilot pulse, sync pulses and data pulses.
- Duration of pause after this block: 2ms (header) or 1s (data).
- note: Do not set FLAG=NONE as for #target tap: In Jupiter-Ace .tap-files the flag byte is actually not written, but in .tzx files it is!

Examples for code blocks with standard timing:

Using the same #code directive as for #target TAP will result in a standard speed data block. So migrating a source from #target TAP to #target TZX is just a matter of changing the #target directive:

```
#CODE name,address,length,flag
```

A data block with some non-standard settings:

```
#CODE name,address,length, FLAG=10, CHECKSUM=NONE, PAUSE=100
```

A data block with a short pilot tone:

```
#CODE name,address,length, FLAG=255, PILOT=999, PAUSE=100
```

Custom pulse lengths and encoding schemes

With #target TZX you can define custom timings for all pulses (pilot, sync and data bits) used for the standard tape encoding scheme and even own schemes, e.g. to write a ZX81-style data block. For this #target TZX introduces new pseudo opcodes to define the timing and encoding scheme of the current #code segment. The .tzx- pseudo opcodes to define pulse length and encoding scheme must be placed immediately after the #code directive.

.tzx-pilot-sym

Defines a sequence of pulses used to construct the pilot tone and the sync pulses. The symbol definition starts with a flag which defines the signal polarity at the start of the symbol and a series of pulse lengths expressed in t-states (3.5MHz). Symbols are indexed in order of appearance.

.tzx-pilot

Uses the above symbols to construct the pilot tone and the sync pulses.

.tzx-data-sym

Defines a sequence of pulses used to encode the data of this #code block. In most encoding schemes one data symbol in the data stream encodes 1 bit of data and you need two different symbols for bit=0 and bit=1, but you can also define an encoding scheme where one data symbol encodes 2 bits, 4 bits or a whole byte. Accordingly there must be either 2, 4, 16 or 256 .tzx-data-sym definitions to encode all possible values. Data symbol definitions use the same syntax as .tzx-pilot-sym.

Polarity flag

Pilot and data symbol definitions start with a polarity flag:

0: Toggle polarity (make an edge, most common case)

1: Don't toggle (make no edge, prolong the previous pulse)

2: Force low level

3: Force high level

Definition of pilot and data symbols

```
.tzx-pilot-sym  polarity, pulse1, pulse2, ...      ; pilot symbol #0
.tzx-pilot-sym  polarity, pulse1, pulse2, ...      ; pilot symbol #1
...
.tzx-pilot      symbol_idx, repetitions, symbol_idx, repetitions, ...

.tzx-data-sym   polarity, pulse1, pulse2, ...      ; data symbol #0
.tzx-data-sym   polarity, pulse1, pulse2, ...      ; data symbol #1
...
; assembler source follows
```

Default values

The symbol tables of a #code block are preset with default values which generate a standard ZX Spectrum (or Jupiter Ace) data block:

```
#code name,address,length,flag=0
    .tzx-pilot-sym  0,2168          ; symbol for pilot pulses
    .tzx-pilot-sym  0,667,735       ; symbol for sync pulses (two pulses)
    .tzx-pilot      0,8063, 1,1     ; 8063 pilot pulses (symbol#0), then one sync
                                      pulse symbol (symbol#1)
    .tzx-data-sym   0,855,855       ; symbol for bit 0
    .tzx-data-sym   0,1710,1710     ; symbol for bit 1

#code name,address,length,flag=255
    .tzx-pilot-sym  0,2168          ; symbol for pilot pulses
    .tzx-pilot-sym  0,667,735       ; symbol for sync pulses (two pulses)
    .tzx-pilot      0,3223, 1,1     ; 3223 pilot pulses (symbol#0), then one sync
                                      pulse symbol (symbol#1)
    .tzx-data-sym   0,855,855       ; symbol for bit 0
    .tzx-data-sym   0,1710,1710     ; symbol for bit 1

#code name,address,length,flag=0, checksum=ace
    .tzx-pilot-sym  0,2011          ; symbol for pilot pulses
    .tzx-pilot-sym  0,601,791       ; symbol for sync pulses (two pulses)
    .tzx-pilot      0,8192, 1,1     ; 8192 pilot pulses (symbol#0), then one sync
                                      pulse symbol (symbol#1)
    .tzx-data-sym   0,795,801       ; symbol for bit 0
   .tzx-data-sym   0,1585,1591     ; symbol for bit 1

#code name,address,length,flag=255, checksum=ace
    .tzx-pilot-sym  0,2011          ; symbol for pilot pulses
    .tzx-pilot-sym  0,601,791       ; symbol for sync pulses (two pulses)
    .tzx-pilot      0,1024, 1,1     ; 1024 pilot pulses (symbol#0), then one sync
                                      pulse symbol (symbol#1)
    .tzx-data-sym   0,795,801       ; symbol for bit 0
    .tzx-data-sym   0,1585,1591     ; symbol for bit 1
```

Choice of tzx block type

The TZX file format provides 4 block types to store code. You can either use a #TZX directive to specify the TZX block to use or use #CODE and leave it to zasm to automatically pick the most suitable block type.

```
#CODE name, address, length, flag
#CODE name, address, length, FLAG=(flag|NONE), [PILOT=(NONE|count)],
                [CHECKSUM=(NONE|ACE)], [LASTBITS=count], [PAUSE=ms]

#TZX STANDARD, name, address, length, FLAG=(flag|NONE), [CHECKSUM=NONE|ACE],
                         [PAUSE=ms]
#TZX TURBO, name, address, length, FLAG=(flag|NONE), [PILOT=count],
                         [CHECKSUM=(NONE|ACE)], [LASTBITS=count], [PAUSE=ms]
#TZX PURE-DATA, name, address, length, FLAG=(flag|NONE), [PILOT=NONE],
                         [CHECKSUM=(NONE|ACE)], [LASTBITS=count], [PAUSE=ms]
#TZX GENERALIZED, name, address, length, FLAG=(flag|NONE), [PILOT=(NONE|count)],
                         [CHECKSUM=(NONE|ACE)], [LASTBITS=count], [PAUSE=ms]
```

The restrictions for each tzx block type are as follows:

0x10: Standard data block:

- LASTBITS are not supported.
- The pulse scheme must not be modified using the .tzx- pseudo opcodes.

0x11: Turbo data block:

- The pulse scheme must match the ZX Spectrum encoding but the pulse lengths (pilot, sync, data bits) and number of pilot pulses can be modified.

0x14: Pure data block:

- PILOT must be NONE
- No pilot must be specified with .tzx-pilot-sym and .tzx-pilot
- The pulse scheme for the data bits must match the ZX Spectrum encoding but the pulse lengths can be modified.

0x19: Generalized data block:

- No restrictions.

Examples for different tzx blocks

Standard speed data block

```
#code name, address, length, flag=255
```

Turbo speed data block

```
#code name, address, length, flag=$ee, pause=150
    .tzx-pilot-sym  0, 900          ; symbol#0 for pilot pulses
    .tzx-pilot-sym  0, 300,400      ; symbol#1 for sync pulses (two pulses)
    .tzx-pilot      0,1520, 1,1     ; 1520 pilot pulses (symbol#0), then one sync
                                      pulse symbol (symbol#1)
    .tzx-data-sym   0, 290,290      ; symbol#0 for bit 0
    .tzx-data-sym   0, 580,580      ; symbol#1 for bit 1
```

Pure data block

```
#code name, address, length, flag=255, pilot=NONE
```

Generalized data block

```
; Example with 2-bit symbols. this requires 4 data symbols for the 4 possible
values:
#code name, address, length, flag=255
    .tzx-pilot-sym  0, 1500             ; symbol#0 for pilot pulses
    .tzx-pilot-sym  0, 500              ; symbol#1 for sync pulses
    .tzx-pilot      0,1000, 1,2         ; 1000 pilot pulses (symbol#0), 2 short
                                          sync pulses (symbol#1)
    .tzx-data-sym   0, 500,350,650,500  ; symbol#0 for 2 bits = 00
    .tzx-data-sym   0, 500,450,550,500  ; symbol#1 for 2 bits = 01
    .tzx-data-sym   0, 500,550,450,500  ; symbol#2 for 2 bits = 10
    .tzx-data-sym   0, 500,650,350,500  ; symbol#3 for 2 bits = 11

; ZX81 program:

#code name, address, length, flag=NONE, checksum=NONE, pilot=NONE
    .tzx-data-sym   3, 530,520,530,520,530,520,530, 4689
    .tzx-data-sym   3, 530,520,530,520,530,520,530, 520, 530, 520, 530, 520, 530,
                       520,530,520,530,4689
```

0x10: Standard speed data block

```
#TZX STANDARD, name, address, length, FLAG=(flag|NONE), [CHECKSUM=NONE|ACE],
[PAUSE=ms]
```

This defines a #code segment which is stored as a standard ZX Spectrum tape block. This segment can be re-entered using directive #CODE or pseudo opcode .AREA. Any following #CODE segments with no FLAG are appended to this segment to form the full tzx data block.

FLAG: The value for the flag byte is in range 0 to 255. If the keyword NONE is used then no flag byte is stored in the tape file.

CHECKSUM: If set to NONE then zasm will append no checksum after the data. If set to ACE then a Jupiter Ace checksum will be calculated. The default is to append the standard ZX Spectrum checksum.

PAUSE: Length of the silent gap after this block in ms. If PAUSE is omitted then the default is 1000ms (header) or 2000ms (data).

0x11: Turbo speed data block

```
#TZX TURBO, name, address, length, FLAG=(flag|NONE), [PILOT=count], [CHECKSUM=NONE|
ACE], [LASTBITS=count], [PAUSE=ms]
```

This defines a #code segment which is stored with the standard ZX Spectrum tape encoding scheme but with different timings. The timings can be set with .tzx- pseudo opcodes as described above. This segment can be re-entered using directive #CODE or pseudo opcode .AREA. Any following #CODE segments with no FLAG are combined with this segment to form the full tzx code block.

FLAG: The value for the flag byte is in range 0 to 255. If the keyword NONE is used then no flag byte is stored in the tape file.

PILOT: Set the number of pilot pulses to the given number. This overrides the default of 8063 or 3223 pulses.

CHECKSUM: If set to NONE then zasm will append no checksum after the data. If set to ACE then a Jupiter Ace checksum will be calculated. The default is to append the standard ZX Spectrum checksum.

LASTBITS: The number of bits actually played from the last byte. Possible values are from 1 to 8. The upper bits of the last byte are used, because the bits were sent from msb to lsb by the ZX Spectrum tape routine. Use of this option is incompatible with a checksum. If LASTBITS is omitted, then the last byte is stored (and played) as usual.

PAUSE: Length of the silent gap after this block in ms. If PAUSE is omitted then the default is 1000ms (header) or 2000ms (data).

Example: Turbo speed block

```
#tzx turbo, my_name, my_address, my_length, flag=$aa, pause=150
    .tzx-pilot-sym  0, 900          ; symbol#0 for pilot pulses
    .tzx-pilot-sym  0, 300,400      ; symbol#1 for sync pulses (two pulses)
    .tzx-pilot      0,1520, 1,1     ; 1520 pilot pulses (symbol#0), then one sync
                                      pulse symbol (symbol#1)
    .tzx-data-sym   0, 290,290      ; symbol#0 for bit 0
    .tzx-data-sym   0, 580,580      ; symbol#1 for bit 1
    ;
    ; machine code follows
```

0x12: Pure tone

Most #tzx directives create TZX file blocks which do not encode a #code block. You can either use the symbolic name of a TZX file block, e.g. PURE-TONE or the hexadecimal number, e.g. 0x12.

```
#tzx PURE-TONE, [COUNT=]num_pulses, [PULSE=]cc_per_pulse
```

TZX block 0x12 can be used to "manually" create a pilot tone. The pulse length is measured in ZX Spectrum clock cycles. (3.5MHz)

COUNT: Number of pulses

PULSE: Length of each pulse expressed in T-states.

0x13: Sequence of pulses of various lengths

```
#tzx PULSES
    dw  NN, ...
```

This block stores up to 255 individual pulses. It can be used to create custom sync pulses after block 0x12 with a custom pilot tone. The pulse length is based on ZX Spectrum clock cycles. (3.5MHz)

0x14: Pure data block

```
#TZX PURE-DATA, name, address, length, FLAG=(flag|NONE), [CHECKSUM=NONE|ACE],
[LASTBITS=count], [PAUSE=ms]
```

This defines a #code segment which is stored with the standard ZX Spectrum tape encoding scheme but with no pilot and sync pulses and optionally with different timing for the data pulses. The data pulse lengths can be set with the .tzx-data-sym pseudo opcode as described above. This segment can be re-entered using directive #CODE or pseudo opcode .AREA. Following #CODE segments without FLAG are appended to this segment to form a full tzx data block.

FLAG: The value for the flag byte is in range 0 to 255. If the keyword NONE is used then no

flag byte is stored in the tape file.

CHECKSUM: If set to NONE then zasm will append no checksum after the data. If set to ACE then a Jupiter Ace checksum will be calculated. The default is to append the standard ZX Spectrum checksum.

LASTBITS: The number of bits actually played from the last byte. Possible values are from 1 to 8. The upper bits of the last byte are used, because the bits were sent from msb to lsb by the ZX Spectrum tape routine. Use of this option is incompatible with a checksum. If LASTBITS is omitted, then the last byte is stored (and played) as usual.

PAUSE: Length of the silent gap after this block in ms. If PAUSE is omitted then the default is 1000ms (header) or 2000ms (data).

0x18: CSW recording block

```
#tzx CSW, [FILE=]"audio.wav", [COMPRESSED], [PAUSE=pause],
          [SAMPLE-RATE=value], [CHANNELS=1|2], [MONO], [STEREO], [SAMPLE-FORMAT=s1|
          u1|s2|u2|s2x|u2x],
          [HEADER=bytes], [START=frame], [END=frame], [COUNT=frames]
```

This block inserts audio data reduced to 1 bit into the .tzx file. The data is RLE-encoded as a so-called 'compressed square wave' and optionally further compressed with Z-compression.

Currently wav files and raw audio files are supported:

- wav files: 1, 2 and 4 byte PCM, 4 byte FLOAT, A-LAW and µLAW encoding, 1 or 2 channels and any sample rate.
- raw files: 1 or 2 bytes, signed or unsigned, little or big endian samples, 1 or 2 channels and any sample rate.

COMPRESSED: Additionally compress the block using Z-compression.

PAUSE: Length of the gap of silence after this block in ms. If PAUSE is omitted then there is no pause after this block.

Raw audio settings

Decoding raw audio files requires some informations:

SAMPLE-RATE: Sample rate of the audio data in samples per second. (STEREO: frames per second.)

CHANNELS: Number of (interleaved) audio channels: 1 or 2. MONO is a short-hand for 1 channel and STEREO is a short-hand for 2 channels.

SAMPLE-FORMAT: The sample format defines 3 aspects of each sample:

s|u: samples are signed (s) or unsigned (u)

1|2: samples are 1 byte or 2 bytes in size.

x: if samples are 2 bytes, then the default is that they are in "network byte order" which means: MSB first. If they are LSB first (as usual for .wav files) an 'x' must be appended.

HEADER: optional: size of a file header. The header will be skipped before extracting audio data or measuring the file positions for START, END and COUNT.

Limit the range to encode

If you don't want to import the whole file then you can adjust the range using START, END and COUNT, measured in FRAMES after the HEADER. If the value for END or COUNT exceeds the end of the file then it is automatically adjusted.

A FRAME is a set of one sample for each channel: 1 sample for MONO and 2 samples for STEREO.

E.g. the size of a frame of 2-byte samples in mono is 2*1=2 and of 4-byte float samples in stereo it is 4*2=8.

Note

The CSW block does not define the starting pulse level, therefore a CSW block might be replayed inverted. To correct this problem zasm always stores a 0x2B set signal level block before the CSW block.

0x19: Generalized data block

```
#TZX GENERALIZED, name, address, length, FLAG=(flag|NONE), [PILOT=(NONE|count)],
[CHECKSUM=NONE|ACE],  [LASTBITS=lastbits], [PAUSE=pause]
```

This defines a #code segment which is stored with a custom encoding scheme. The encoding scheme and pulse timings can be set with .tzx- pseudo opcodes as described above. This segment can be re-entered using directive #CODE or pseudo opcode .AREA. Following #CODE segments without FLAG are appended to this segment to form a full tzx data block.

FLAG: The value for the flag byte is in range 0 to 255. If the keyword NONE is used then no

flag byte is stored in the tape file.

PILOT: The number of pilot pulses is set to the given number. This overrides the default of 8063 or 3223 pulses.

CHECKSUM: If set to NONE then zasm will append no checksum after the data. If set to ACE then a Jupiter Ace checksum will be calculated. The default is to append the standard ZX Spectrum checksum.

LASTBITS: The number of bits actually played from the last byte. Possible values are from 1 to 8. The upper bits of the last byte are used, because the bits were sent from msb to lsb by the ZX Spectrum tape routine. Use of this option is incompatible with a checksum. If LASTBITS is omitted, then the last byte is stored (and played) as usual. If 2- or 4-bit data symbols are used (if 4 or 16 data symbols were defined) then LASTBITS must be a multiple of 2 or 4. If 8-bit data symbols are used (if 256 data symbol were defined) then LASTBITS must be a multiple of 8 which means, yes, it must be 8.

PAUSE: Length of the silent gap after this block in ms. If PAUSE is omitted then the default is 1000ms (header) or 2000ms (data).

Example: Turbo speed block

```
#tzx generalized, my_name, my_address, my_length, flag=$ee, pause=150
    .tzx-pilot-sym  0, 900          ; symbol#0 for pilot pulses
    .tzx-pilot-sym  0, 300,400      ; symbol#1 for sync pulses (two pulses)
    .tzx-pilot      0,1520, 1,1     ; 1520 pilot pulses (symbol#0), then one sync
                                      pulse symbol (symbol#1)
    .tzx-data-sym   0, 290,290      ; symbol#0 for bit 0
    .tzx-data-sym   0, 580,580      ; symbol#1 for bit 1
    ;
    ; machine code follows
```

0x20: Pause (silence) or 'Stop the tape' command

```
#tzx PAUSE, [DURATION=]pause
```

Insert silence into the tape.

DURATION: duration in ms. If PAUSE=0 then an emulator will stop the tape.

0x21: Group start

```
#tzx GROUP-START, [NAME=]name
```

Organize some blocks into a group with a name. Used to group bleepload blocks or the like. Must be followed by tzx group-end

NAME: name to display.

0x22: Group end

```
#tzx GROUP-END
```

End of group.

0x24: Loop start

```
#tzx LOOP-START, [REPETITIONS=]repetitions
```

Start a group of blocks to repeat. Mostly used in old TZX files. Must be followed by #tzx loop-end.

REPETITIONS: number of repetitions. Must be greater than 1.

0x25: Loop end

```
#tzx LOOP-END
```

End of group.

0x2A: Stop the tape if in 48K mode

```
#tzx STOP-48K
```

This block instructs an emulator to stop the tape only if it is currently emulating a 48k (or 16k) model. Else the tape is left running and the 128k model can load additional data.

0x2B: Set signal level

```
#tzx POLARITY, [POLARITY=]polarity
```

Set the signal polarity. The ZX Spectrum tape loading routines are polarity insensitive but e.g. the ZX81 routines weren't. This block can be used to force the polarity of the following pulses.

POLARITY: 0=low, 1=high.

0x30: Text description

```
#tzx INFO, [TEXT=]text
```

Add a text description for the following block(s). It is up to the emulator to display it somehow.

TEXT: Ascii text. Should be limited to 30 characters.

0x31: Message block

```
#tzx MESSAGE, [DURATION=]duration, [TEXT=]text
```

Display a message during loading.

DURATION: time to display this message in seconds.

TEXT: Ascii text. At most 8 lines à 30 characters. Text lines must be enclosed in quotes and lines separated by comma.

Example

```
#tzx message, 5, "Hello world,","this is a test message.","cheese."
```

0x32: Archive info

```
#tzx ARCHIVE-INFO
    db  type, text
    ...
```

Store information like author, publisher, release date, price etc.

Only defb pseudo opcodes (and aliases) are allowed. Each line must consist of one byte indicating the type of information and a quoted text string giving this information.

There must be at most one ARCHIVE-INFO block for each tzx file. The archive block is reordered to the front of the file regardless of it's position in the source.

Example

```
#tzx archive-info
    db  0,"Fufu goes bobo"      ; title
```

```
        db  3,"2018"                ; year
        ...
```

Information IDs (tzx v1.20):

00 - Full title

01 - Software house/publisher

02 - Author(s)

03 - Year of publication

04 - Language

05 - Game/utility type

06 - Price

07 - Protection scheme/loader

08 - Origin

FF - Comment(s)

0x33: Hardware type

```
    #tzx HARDWARE-INFO
        db  type, id, state
        ...
```

Store information about supported or required computer models and peripherals. For the full list refer to the TZX file documentation.

The contained data should parse to multiple sets of three bytes, 'hardware type', 'hardware ID' and 'hardware info'. The data may be stored with db or similar as you like.

There must be at most one HARDWARE-INFO block for each tzx file. The hardware-info block is reordered to the front of the file (behind the archive info, if present) regardless of it's position in the source.

Example

```
    #tzx hardware-info
        db  0,0x1a,3            ; doesn't run on a jupiter ace
        db  0,0x00,0            ; game is known to run on a ZX Spectrum 16k
        db  2,0x01,3            ; doesn't run if a MF1 is attached
        db  3,0x02,1            ; actually uses the Currah µSpeech
```

#target .O and .80

this format is not yet well tested. If there are problems please send an email to Kio.

This creates a tape file for use with an ZX80 emulator. The '.o' and '.80' file formats are identical. The tape file will be loaded to the ram start at address $4000. There was no choice…

Code may be stored in one or more #code segments. However, the first segment must be at least 0x28 bytes long and contain the system variables.

```
    #target 80
    #code   <name>,<start>,<len>
        ...
```

See the ZX80 .o template file: template_o.asm

The tape data is always loaded to $4000, so this is the only choice for the <start> address in the

#code directive. The <size> may be any value up to a maximum of $C000, but most ZX80 had at most 16 kB of ram if extended with an external memory expansion, only 1 kB if none.

The ram starts with system variables from $4000 to $4028, which must be set to proper values. The variable E_LINE at $400A, which contains the end of ram address, must be calculated properly and is checked by the assembler.

Note that zasm has a convenient assembler directive to translate from the ascii (or utf-8) characters in your source file to the non-ascii character set of the ZX80:

```
    #charset ZX80
```

You can include c sources in your assembler file, but there are two pitfalls:

You can use the character set translation of zasm to translate strings in your c source as well. But unluckily character literals are not exported as character literals by the c compiler sdcc but as their ascii code and zasm has no chance to detect this. So character literals in your c source are not translated.

Second, character 0x00 is used in c sources as a string end indicator, but it is also a valid character for the ZX80: the space. So you'll have to think of a way to work around this problem.

#target .P, .81 and .P81

this format is not yet well tested.

If there are problems please send an email to Kio.

This creates a tape file for use with a ZX81 emulator. The tape file will be loaded to the ram start + 9 at address $4009.

"p" can be used instead of "81". This will create a target file with extension "p" or "81" respectively, which are fully identical. ".p81" files also include the program name which must be written using the ZX81 character set.

Code may be stored in one or more #code segments. However, the first segment must be at least 0x403B - 0x4009 bytes long and contain the system variables. (except for the first 9 bytes which are not saved.)

```
    #target 81
    #charset ZX81
    #code <name>,<start>,<len>
        …
```

or:

```
    #target p81
    #charset ZX81
    #code   _PROGNAME
```

```
        dm  "progname" | $80        ; zasm converts the characters to charset ZX81!
    #code   _RAM, 0x4009, _ram_end - 0x4009
        …
```

See the ZX81 .p template file: template_p.asm

The tape data is always loaded to $4009, so this is the only choice for the <start> address in the #code directive. The <size> may be any value up to a maximum of $C000 minus $09, but most ZX81 had at most 16 kB ram if extended with an external memory expansion, only 1 kB if none. (The TS1000 had whoopy 2k!)

The ram starts with system variables from $4009 to $403C, which must be set to proper values. The variable E_LINE at the address $4014, which contains the end of ram address, is checked by the assembler.

This file also contains the screen file, which basically means, that the video memory must contain 25 HALT opcodes if the file was saved in SLOW mode.

Here are few **important variables**:

```
$4014   defines the
end address (used to calculate the file length)
$4029   points to the next executed (autostarted) BASIC line
$403B   bit 6 indicates
if program runs in SLOW or FAST mode
$403C++ may be misused for whatever purpose
```

zasm has a convenient assembler directive to translate from the ascii (or utf-8) characters in your source file to the non-ascii character

set of the ZX81:

```
#charset ZX81
```

You can include c sources in your assembler file, but there are two pitfalls:

You can use the character set translation of zasm to translate strings in your c source as well. But unluckily character literals are not exported as character literals by the c compiler sdcc but as their ascii code and zasm has no chance to detect this. So character literals in your c source are not translated.

Second, character 0x00 is used in c sources as a string end indicator, but it is also a valid character for the ZX81: the space. So you'll have to think of a way to work around this problem.

#target ACE

```
#target ACE
#code VRAM_COPY,   $2000, $400
#code VRAM,        $2400, $400
#code CRAM_COPY,   $2800, $400
#code CRAM,        $2C00, $400
#code RAM_COPIES,  $3000, $C00
#code SYSVARS,     $3C00, $40
#code RAM,         $3C40, ramsize - $840
```

This target creates a snapshot file for use with a Jupiter Ace emulator.

This format may be reworked to remove the necessity to include the empty mirror pages

The ram of the Jupiter Ace starts with several copies of the video ram and the character ram, before the usable area at address $3C00 is reached.

The Z80 registers and some settings are stored in the first page at $2000. Since all system variable locations are included in this file, there is probably no need to define a #data segment, except if you include c sources.

Allowed ram sizes are 3k, 3+16=19k and 3+32=35k. This results in a total of $2000, $6000 or $A000 bytes in all code segments, including mirrors.

See the Jupiter Ace .ace template file: template_ace.asm

Basic layout of the ace file:

```
#target ACE
#code VRAM_COPY,   $2000, $400
;
; this is a "copy" of the video ram at $2400.
; it should be empty except for the
Z80 registers which are stuffed in here.
; see the template file for the layout of this data.
;
#code VRAM,        $2400, $400
;
; this is the actually loaded data for the video ram.
; you may put here a greeting message (in ascii) which is instantaneous visible.
;
#code CRAM_COPY,   $2800, $400
;
; a copy of the character ram which follows. must be empty.
;
#code CRAM,        $2C00, $400
;
; the character ram.
; simply #insert the file "
Examples/jupiter_ace_character_ram.bin" here.
;
#code RAM_COPIES,  $3000, $C00
;
; 3 copies of the ram at $3C00. must be empty
;
#code SYSVARS,     $3C00, $40
;
; the Forth system variables. must be set up properly.
; see the template file for the layout. For the values ask someone else... :-|
;
If you can give some advice or a sample setup please send email to Kio. Thanks!
;
if you don't use the Forth rom then you may use the whole rom as you like.
;
#code RAM,         $3C40, ramsize - $840
;
; the free ram after the system variables.
```

List File

zasm generates a list file, except if command line option -l0 is given.

Option '-u' includes the generated opcode in the list file.

Option '-w' enables a labels listing at the end of the list file.

Option '-y' includes accumulated cpu cycles in the list file.

Plain listing

```
NMI:    push af
        push hl
        ld   hl,($5cb0)

        LD   A,H               ;falls HL=0, dann Kaltstart
        OR   L
        JR   NZ,M0070          ;sonst passiert nichts
        JP   (HL)
M0070:  POP  HL
        POP  AF
        RETN

NEXZEI: LD   HL,(CHADD)        ;Programmzeiger erhoehen
M0077:       INC HL
M0078:       LD (CHADD),HL     ;Adr. des naechsten zu
                               ;interpret. Zeichens
        LD   A,(HL)            ;neues Zeichen laden
        RET
```

Listing with object code

```
0131: 110100    calcspeed:  ld   de,1       ; pre-adjust
0134: 210040                ld   hl,tickercell
0137: 76                    halt
0138: 7E                    ld   a,(hl)
0139: 0605      cs1         ld   b,5        ; 7
013B: 05        cs2         dec  b          ; 5*4
013C: C23B01                jp   nz,cs2     ; 5*10
013F: 13                    inc  de         ; 6
0140: BE                    cp   a,(hl)     ; 7
0141: CA3901                jp   z,cs1      ; 12
0144: EB                    ex   hl,de
0145:           ; this took hl*100 ticks for 1/60 sec
0145:           ;  = hl*6,000 ticks for 1 sec
0145: 110600                ld   de,6
0148: CDB000                call mult
014B:           ;  = hl*1,000 ticks for 1 sec
014B: 11E803                ld   de,1000
014E: CDC900                call divide
0151:           ;  = hl*1,000,000 + de*1,000 ticks
0151: C9                    ret
```

Listing with object code and cpu cycles

Cpu cycles are accumulated from the last label position. Branching opcodes are given with their run-through time and the branching time. Opcodes like LDIR also show the time for a loop.

This example is taken from a file generated by sdcc, so don't mind the unusual syntax for immediate arguments and local labels.

```
67A3:                   00102$:
67A3:                   ;/Develop/Projects/zasm/lib/___fsgt.c:71:
if (fl1.l<0 && fl2.l<0) {
67A3: 210400   [10]         ld  hl,#0x0004
67A6: 39       [21]         add hl,sp
67A7: 56       [28]         ld  d,(hl)
67A8: 23       [34]         inc hl
67A9: 5E       [41]         ld  e,(hl)
67AA: 23       [47]         inc hl
67AB: 4E       [54]         ld  c,(hl)
67AC: 23       [60]         inc hl
67AD: 66       [67]         ld  h,(hl)
67AE: CB7C     [75]         bit 7, h
67B0: 2849     [82|87]      jr  Z,00106$
67B2: 210000   [92]         ld  hl,#0x0000
67B5: 39       [103]        add hl,sp
67B6: 56       [110]        ld  d,(hl)
67B7: 23       [116]        inc hl
67B8: 5E       [123]        ld  e,(hl)
67B9: 23       [129]        inc hl
67BA: 4E       [136]        ld  c,(hl)
67BB: 23       [142]        inc hl
67BC: 66       [149]        ld  h,(hl)
67BD: CB7C     [157]        bit 7, h
67BF: 283A     [164|169]    jr  Z,00106$
```

Label listing

This section lists the defined code and data segments and the global and all local labels (if any).

For each label zasm lists the value or whether it is invalid (not defined), the segment it is defined in, the source file it is defined in and the source line and whether it is unused.

In this example no code segment was defined and all labels reside in the default segment which is currently named "(DEFAULT)" which may look a little bit odd.

```
; +++ segments +++

#CODE (DEFAULT): start=0     len=1684

; +++ global symbols +++

hd64180               = $0001 =      1  (DEFAULT) :1 (unused)
loop2                 = $0249 =    585  (DEFAULT) test-opcodes.asm:636
n                     = $0040 =     64  (DEFAULT) test-opcodes.asm:593
n1                    = $0001 =      1  (DEFAULT) test-opcodes.asm:1357
n16                   = $0010 =     16  (DEFAULT) test-opcodes.asm:1358
n6                    = $0006 =      6  (DEFAULT) test-opcodes.asm:1356
nn                    = $4142 =  16706  (DEFAULT) test-opcodes.asm:592
test_addressing_modes = $0001 =      1  (DEFAULT) test-opcodes.asm:14
test_compound_opcodes = $0001 =      1  (DEFAULT) test-opcodes.asm:15
test_expressions      = $0001 =      1  (DEFAULT) test-opcodes.asm:13
test_fails            = $0001 =      1  (DEFAULT) test-opcodes.asm:16

; +++

local symbols +++

anton   = $0014 =     20  (DEFAULT) test-opcodes.asm:41 (unused)
n20     = $0014 =     20  (DEFAULT) test-opcodes.asm:37
n5      = $0005 =      5  (DEFAULT) test-opcodes.asm:36
```

```
; +++

local symbols +++

(DEFAULT)$1 = $0055 =     85  (DEFAULT) test-opcodes.asm:229
n6          = $0006 =      6  (DEFAULT) test-opcodes.asm:262
nn          = $0040 =     64  (DEFAULT) test-opcodes.asm:261
```

Errors

Errors are marked with ***ERROR*** in the list file. The assembler will abort if too many errors occur.

source file:

```
#target rom
#code ROM,*,1000
    db  foo
#end
```

command line and output:

```
$> zasm zzz.asm

 in file zzz.asm:
3:
db  foo
           ^ label "foo" not found
assemble: 6 lines
time: 0.0015 sec.

zasm: 1 error

$>
```

list file:

```
#target rom
#code ROM,*,1000
    db  foo
           ^ ***ERROR***

label "foo" not found
#end
```

If you have only a single source file and you encounter only a few errors and you didn't enable additional fields in the list file then you can move the source into the trash, rename the list file and work on it as your new source file. You can easily find all the ***ERROR***s and correct them, then start over again. But be careful that the list file is not truncated before actually deleting it.

If you suspect the assembler is generating wrong object code, you can include the object code in the listing. Then you can easily verify the generated code.

If there really is a bug, send a bug report to Kio.

Legal and Version History

Copyright

Zasm Z80 assembler, copyright © 1994-2020 Günter Woigk, kio@little-bat.de

Permissions

Permission to use, copy, modify, distribute, and sell this software and its documentation for any purpose is hereby granted without fee, provided that the above copyright notice appears in all copies and that both that copyright notice, this permission notice and the following disclaimer appear in supporting documentation.

Disclaimer of Warranties

THIS SOFTWARE IS PROVIDED "AS IS", WITHOUT ANY WARRANTY, NOT EVEN THE IMPLIED WARRANTY OF MERCHANTABILITY OR FITNESS FOR A PARTICULAR PURPOSE AND IN NO EVENT SHALL THE COPYRIGHT HOLDER BE LIABLE FOR ANY DAMAGES ARISING FROM THE USE OF THIS SOFTWARE, TO THE EXTENT PERMITTED BY APPLICABLE LAW.

Version history

```
1.0.0  1996: First version for private use. Mac OS 7.0. No public release.
2.0.0  2000: Total rework with new libraries.
2.0.7  2002: started port of MacOS classic version to unix
3.0.0  2002: MacOS X command line version released
3.0.2        .tap and .sna support implemented
3.0.13 2005: #code statement now also sets the physical address for intel hex files
3.0.14       #target, #end and #code now optional. default target = rom
3.0.15       added illegals with XL,XH,YL,YH to: cp, or, ld, and, xor, sub, sbc,
             adc
4.0.0  2014: rewrite with more C++ inside
4.0.0  2014: .tap .sna .z80 .o .p .ace
4.0.0  2014: c compiler support
4.0.0  2014: rework of segment handling
4.0.0  2014: #local #endlocal
4.0.0  2014: #charset
4.0.0  2014: #assert and directive '!' for self test
4.0.0  2014: 8080 and HD64180 support
4.0.0  2014: list accumulated cpu cycles
4.0.1  2014: write Motorola S-Record files
4.0.2  2015: added support for native 8080 assembler source
4.0.3  2015: added more support for alternate/various/weird syntax
4.0.4  2015: added macro and rept, .phase and .dephase
4.0.5  2015: #define, test suite, --flatops, Linux version
4.0.7  2015: "extended arguments" in macros with '<' … '>'
4.0.8  2015: fixed bug in .ACE file export
4.0.9  2015: fixed bug in .81 export, secure cgi mode
4.0.10 2015: bug fixes, added Z80 instructions in 8080 assembler syntax
4.0.11 2015: Made Linux happy again
4.0.16 2016: FreeBSD version
4.0.18 2016: illegals: allow 'ixh' … 'iyl' for index register halfes
4.0.19 2016: included TextMate bundle, added #cpath to set c-compiler
             path in source
```

```
4.0.20 2017: minor rework of c-compiler handling, bug fixes
4.0.21 2017: allow multiple opcodes per line after '\'
4.0.24 2017: define NAME_size and NAME_end labels for all segments
4.1.0  2017: included Einar Saukas' ZX7 "optimal" LZ77 compressor
4.1.2  2017: fixed problems with compressed size validity
4.1.3  2017: new function sin() and cos() to easily build wave tables
4.1.4  2017: added pseudo instructions DUP and EDUP as an alias for
             REPT and ENDM
4.1.5  2017: reworked the regression test framework. support for '#!' in line 1 of
             the source file
4.2.0  2018: new #target TZX: directly write to .tzx tape files
4.2.1  2018: Bug fixes. This was to be expected.
4.2.2  2018: Fixed bug where the binary file was appended to the hex or s19 file
4.2.3  2018: Fixed bug where error messages were garbled
4.2.4  2019: Fixed bug in .rept/.endm or .dup/.edup sanity check
4.2.6  2020: added text replacement in macros for values between '{' and '}'.
4.2.7  2020: hexfile for #target BIN now uses cpu addresses for ram loaders
4.2.8  2020: cmd line option "--date" to set reproducible date and time
4.2.9  2020: fixed cpu cycle calculation for Z80180
4.3.0  2020: new option --convert8080
4.3.1  2020: fixed casefolding bug in macro argument evaluation
4.3.4  2020: added &1234 syntax for hex numbers
4.3.5  2020: fixed bug with trailing 0x00/0xFF bytes not written to .hex and .s19
             file
4.3.6  2020: 'org' now can have a 2nd arg for the fill byte. fixed Z180 cycle
             counting, shebang.
4.4.0  2020: built-in testcode runner for automated tests
```