ZASMB Documentation

"zasmb"
A Zilog mnemonic assembler
(C) P.F Ridler 1984

P.F Ridler,
4, Lewisam Ave.,
Chisipite,
Zimbabwe.

0.0 **Contents**.

0.1 **Introduction**.

"zasmb" is a vanilla flavoured assembler for machines based on the Z80 microprocessor. It does not produce relocatable code, it does not handle macro-instructions and it does not have any fancy operators. It does have the capability to read source statements from more than one file, it does have conditional assembly facilities and it is both fast and cheap. It has been used by its author over several years for very simple jobs and for assembling complex programs such as compilers and has not been found wanting.

The main requirement for producing relocatable code goes this way. "I have a very large program which I am writing in parts. Why should I constantly reassemble those parts which are are already written and tested? If they are partially assembled in relocatable form then I can save time by just linking them into the piece of the program which I am currently testing".

There are some very valid counter arguments. Assemblers are no longer as slow as they were. The time "zasmb" takes to assemble a source program of 3500 lines to an executable code file is about 30 seconds using a ram disc emulator. There is no linking time at all. To try to save time over this sort of performance is hardly worthwhile.

When a program is in the process of being written in parts it is very seldom that one part is really finished and tested; the process is iterative and usually a major change in one part will involve minor changes to other parts, which then have to be reassembled. There are such routines as those which display a message or open a file, which will not require amendment but these should be stored in a library, and this can be done whether they are in relocatable or source code form.

The matter of macro-instructions is another matter altogether. There are people who are very fond of macros: the author is not one of them. Their main use is for the development of pseudo-languages, and for this purpose they are vital if no compiler-compiler is available. They can also save a little typeing effort if several similar subroutines are used in the same program, but here there is a danger in that the programmer may try to tailor the program to suit the available macros rather than writing code to suit the problem which the program is to solve.

"zasmb" is sufficiently fast that relocatable code offers little or no advantage in speed and it can, by using an "include" statement, use source code segments from a library.

"zasmb" is an assembler for the Zilog/Mostek Z80-CPU microprocessor. It is designed to run under the CP/M operating system from Digital Research. On a Z80 microprocessor based system. "zasmb" will run in a 32K CP/M system but will accommodate larger programs in more memory up to the maximum of 64k bytes addressable by a Z80 system.

"zasmb" assembles at over 7000 lines per minute using a 4MHz Z80-based machine with a solid state disc emulator. It could probably be made to perform faster if the source input routines were improved, but it is thought that the improvement which might be obtained would not justify the effort involved.

"zasmb" reads a source (.Z80) file produced by a text editor and produces an object code (.COM) file. and an optional listing (.LST) file.

1. Getting started.

To use "zasmb" a source file having a file type (extension) ".Z80" must be prepared in Zilog mnemonics and this is assembled by issuing the CP/M command

```
<d>:zasmb : <name>
```

where <d> is the letter of drive on which the source file resides and is the name of the .Z80 source file.

The input file must have an extension of ".Z80" and the output file will have an extension of ".COM" if it is created. A .COM file will not be produced if there are errors signalled during assembly. A .LST file will be created if the "list on" statement has been included in the program irrespective of whether there are errors in the program or not. Even if a listing is not requested any errors that occur will be listed.

2. Assembly language.

The assembly language mnemonics used by "zasmb" are those in the Zilog manual.

2.1 Character set.

The character set recognised by the assembler consists of the letters:

```
                  abcdefghijklmnopqrstuvwxyz
                  ABCDEFGHIJKLMNOPQRSTUVWXYZ
                  @_
 the digits:      0123456789
 and the special characters:   + - |  * / &
```

The assembler does not distinguish between upper and

lower case versions of the same character except within text strings.
Text strings may contain any of the printable characters in the ASCII set.

2.2 **Statements.**

The form of an input statement is:

```
    [label[:]] [operator [expression [,expression]]] [;[text]]
```

where the fields in braces are optional. If all the optional fields are omitted a blank line results, which is acceptable to the assembler.
Between the fields any number of blank or tab characters may be present but within fields other than there may be no or characters.
The length of an input statement is limited to 80 characters.
The assembler recognises four fields in a statement (line). These are:

labels,
operators,
operands,
comments.

Any two fields must be separated from each other by at least one space or character.

2.2.1 **Labels.**

Labels must start in the first column of a line and may consist of up to 7 characters. (The number of characters in a label may be increased by altering the constant "mxnmch" and reassembling.) The first character of a name must be a letter but subsequent characters may be either letters or digits.
The symbols "_" and "@" are regarded as letters and may be included in labels.
Upper case and lower case letters are interpreted as being identical except when included in text strings.
A label must start with a letter in column 0. It may consists of from 1 to 7 characters and digits, and, optionally, it may be followded by a colon, with no space between the last character and the colon. The maximum number of characters was chosen to be compatible with the normal tabulation columns 0,8,16 etc. and is not a fundamental limitation in the assembler program. A few minor changes to the progam would allow names to be longer.
The following labels are reserved for the Z80 registers:

```
                a,b,c,d,e,f,h,l,i,r,af,bc,de,hl,ix,iy,sp
```

2.2.2 **Operators.**

The operator, if present, must be separated from the label (or ":") by at least one or character. The operators are those from the Zilog mnemonic set together with some pseudo-operators which are in common use.
The complete set of operators is:

```
                add adc and
                bit
                call ccf cp cpl cpi cpir cpd cpdr
                daa dec di djnz
                ei ex exx
                halt
                im0 im1 im2
                in ini inir ind indr inc
                jp jr ld ldi ldir ldd lddr
                neg nop
```

```
                  or out outi otir outd otd
                  push pop
                  res ret reti retn rst rlca rla rrca
                  rra rlc rl rrc rr rld rrd
                  scf set sub sbc sla sra srl
                  xor
```

The Z80 manual gives the forms

```
                  or a
                  cp const
```

which are inconsistent with the form

```
                  add a,b
```

"zasmb" allows the forms

```
                  or a,a
                  cp a,const
```

for the sake of consistency.

The manual is also unclear whether the interrupt mode command should be

```
                  im 0 or im0
```

so that the forms "im0", "im1" and "im2" have been used because they are easier to implement.

2.2.2.1 Pseudo-operators.

The set of processor operators is augmented by the assembler instuctions:

```
                  equ   db         dw       ds       list  include
                  defb  defw       defs     read
                  org   forg       end      if       endif
```

where the forms on the second line are alternatives to those above them.

For a description of the individual pseudo-operators see Section 5.

2.2.3 **Operands.**

The operand field of a statement may contain none, one or two operands. If there is more than one operand, they must be separated by a comma.

Operands comprise one of the following forms:

- a number,
- a label,
- the program counter "$",
- a string,
- an expression composed of the above.

For a description of the syntax of expressions see Section 8.

2.2.4 **Comments.**

A comment consists of a semi-colon folllowed by optional explanatory text. It must be the last (or the only) field on a line, and may not continue beyond the end of the line.

3.0 **Error messages.**

Errors detected during assembly are always displayed on the console and are entered into a .LST file even if the source file does not include a "list on" statement.

The following text is a copy of the list file of a program which has errors in it.

```
     0                                 list on
     1                  ;
     2                  ;
A    3  0100  78                       ld     a,bc
```

```
                                          ^ Argument error
U    4  0101  2A0000                 ld     hl,(abc)
                                             ^ Undefined name
X    5  0104  79                     ld     a,c,
                                          ^ Extra character ","
A    6  0105  61626364               db      'abcdef
              6566
                               ^ Argument error
     7                  ;
     8                  ;
     9  010B
```

Column 0 holds a single letter abbreviation of the error message while underneath the erroneous line there is an arrow pointing to the approximate position of the error followed by a fuller error message. If the "list on" statement were not present in source file, then there a .LST file would still be generated but only those lines which have errors would be listed together with their error messages.

3.1 Error recovery.

When the assembler encounters an error it display on the console, below the error message, the additional message

```
             Edit, Continue or Quit? (E|C|Q)
```

The "C" and "Q" replies are obvious but the "E" is not. If the "Edit" option is exercised the assembler writes into the CP/M CCP buffer the command line

```
             zedit d:name.z80
```

followed by the row and column numbers of the position of the error and then jumps to the base of the CCP, which is not overwritten by the assembler. This invokes the editor and positions the cursor at the error position. This facility must be disabled if the user's editor will not respond to line and column arguments.

Disabling the communication with the editor is done by changing the statement

```
             edit    equ     true
```

near the beginning of "zasmb.z80" to

```
             edit    equ     false
```

The effect will be to make the "E" reply the same as "C"
This feature of the assembler has been found to speed up assembly language program development considerably. For further detail see the file "comments.pfr".

4. **Options.**

There is only one option. This is the Intel "Hex" option. If the command line has the token "hex" ("h" will suffice) following the filename as in

```
             zasmb d:name hex
```

an "Intel hex" code file will be generated instead of an executable .COM file. This file will have the extension .HEX and must be loaded into memory using either "DDT" or "zload".

5.0 **Pseudo-operators.**

"zasmb" has a number of assembler directives or "pseudo--operators. Most of these occur in other assemblers except for "forg" the "false origin" directive. These pseudo-operators are described in detail in the sections following.

5.1 db (or defb)

This pseudo-operator defines a byte or a sequence of bytes to have the values calculated from the list following.

Obviously, each item in the list must have a value in the range 0..FFH and it is left to the programmer to ensure that this is so.

The form of the statement is :

```
        [label] db  [comment]
```

It reserves and initialises one or more bytes to the sequence of values given by the expression list. The label, although usual, is optional and the expression list is one or more expressions separated by commas. There is no limit on the length of the expression list except that imposed by the line length of 80 characters.

A string may be used as a shorthand form of an expression list, when all the items in the list would otherwise be character expressions.

However, while string expressions may be composed of any number of characters, other expressions must evaluate to values which will fit into a one byte storage element.

```
    e.g.  string  db      'abcdef',cr,lf
          junk    db      -10H,+13,2345   ;illegal (2345 too big)
```

5.2 **dw (or defw).**

This pseudo-operator defines a word (two bytes) or words to have the values calculated from the list following. Each item in the list must have a value which will fit into one word but unlike the "db" statement strings are not allowed. It is left to the programmer to ensure that this is so.

The form of the statement is :

```
[label] dw  [comment]
```

It reserves and initialises one or more words to the sequence of values given by the expression list. The label, although usual, is optional and the expression list is one or more expressions separated by commas. There is no limit on the length of the expression list except that imposed by the line length of 80 characters.

5.3 **ds (or defw).**

This pseudo-operator reserves a sequence of memory locations for future use.

The form of the statement is :

```
        [label] ds  [comment]
```

It reserves, but does not initialise, the number of bytes given by the result of the expression. The label, although usual, is optional.

5.4 equ.

This pseudo-operator defines a name (label) to represent a constant value.

```
        label equ  [comment]
```

defines the label to have the (constant) value of the expression following. The expression must evaluate to a value in the range 0..FFFFH so as not to exceed the 2-byte storage space allowed for it in the symbol table. Strings are not allowed.

```
e.g.  lf      equ     10              ;decimal value
      cr      equ     0DH             ;hex value
      string  equ     'abcdef'        ;illegal
```

5.5 **include (or read).**

The "include" statement allows source language statements to be read from a file other than the original source file. It takes the form:

```
        [label]   include :.    [comment]
```

```
or            [label]   read    :.    [comment]
                     where     is a valid disc drive letter,
                            is a valid CP/M file name,
                      and    is a valid CP/M file type.
```

The supplementary source file must obviously contain valid assembly language statements but must not contain an "end" statement unless this is meant to be the last line of the entire program.

The optional label may be included, but it is rather pointless, if not dangerous.

"include" files may be nested to a depth of 4. This can be altered easily.

5.6 **if.**

An "if" statement signals the start of statements which are to be conditionally assembled. It has the form:

```
[label] if expression [comment]
```

The expression should have only the values (=1) or (=0). If it has any value other than 1 it will be taken to have the value .

When the expression is "true" the lines following the "if" statement will be assembled in the normal manner until an "endif" statement is encountered.

If the expression following the "if" has the value the source lines following it will be ignored until an "endif" statement is encountered, after which normal assembly will be resumed. The list file, if any, will contain the ignored lines, but they will have no operation codes entered against them.

5.7 **endif.**

This statment signals the end of a section of the program which is to be conditionally assembled. It takes the form

```
[label] endif [comment]
```

It will always cause normal assembly to resume.

5.8 list

This pseudo-operator takes one or other of the arguments "on" or "off". If the argument is "on", a listing file is created on the disc on which the source file resides. This listing file continues until the end of the assembly or until another "list" operator with the argument "off" is encountered.

At the start of an assembly the list file is considered to be turned "off" and will remain so until a "list on" statement is encountered.

The "list" operator turns the assembler output listing file on or off.

```
[label]    list on       ;turns on the listing file
[label]    list off      ;turns off the listing file
```

The list file has the same name as the assembly language source file but the type (extension) is changed to ".LST". It will thus be on the same disc as the source file.

If the list file is already "on", further "list on" commands will have no effect and similarly for "list off" commands.

When the list file is turned on, a file is created containing the following information:

the number of each line,
the program counter value at the start of the instruction,

the code generated by the assembler for the statement,
the text of the statement itself.

If the statement preceded by the line number etc. is longer than 80 characters it is truncated to fit on an 80 column line. An example of the contents of a list file follows.

```
     1                              list on
     2                   ;
     3        (0000)     label         equ     0
     4        (0001)     true          equ     1
O    5  0100             false         eqq     0
                                         ^ ***** Op-code error *****
     6                   ;
     7        (0001)     debugs        equ     true
     8                   ;
     9                   ;
    10  0100                           org     100H
    11                   ;
    12  0100  (0010)     aaaa          ds      10H
    13  0110                           if      debugs
    14  0110  3E0A                     ld      a,10
    15  0112  61626364                 db      'abcd'
    16  0117  0600                     ld      b,0
    17  0119  3620                     ld      (hl),' '
    18  011B  (0010)     bbbb          ds      10H
    19  012B                           endif
    20  012B  3E00       cccc          ld      a,0
    21                   ;
    22  012D                           end
```

5.9 **org.**

This statement defines the starting address (origin) of any section of the program. It has the form:

```
[label] org expression [comment]
```

and the expression, which must have a value in the range 0..FFFFH gives the starting address of the section of program.

A program may have more than one "org" statement but a new origin may not be less than the current value of the program counter, i.e. the program counter must not be driven backwards.

```
org 100H
 .  .
 .  .
 .  .
org 1000H
 .  .
 .  .
 .  .
org 800H ;illegal, ( < previous pc )
```

5.10 **forg.**

This pseudo-operator defines a "false origin". It allows code to be generated which will be executed at some address other than that at which it is assembled. It was included so that a loader could be included in a version of the CP/M CCP which had its' origin at 100H. The loader was to be included in this file but was to be transferred to E000H before it was executed. If it had had an origin of E000H then the file would have been more than 60k long.

This statement defines the starting address (origin) of a section of the program. It has the form:

```
[label] forg expression [comment]
```

and the expression,

which must have a value in the range 0..FFFFH, gives the starting address of the following section of program.

The "forg" address is nullified by another "org" pseudo-operator.

6.0 **Known bugs.**

Latest revision: 1 Mar 1986
no check on expression size.
some (ix+d) forms are not checked properly.

If further bugs are discovered please notify the author, preferably with the fix, giving the source code producing the bug and as much other relevant information as possible.

7.0 Files on this disc.

z80asmb.z80 the main source code file for the assembler
z80asmb.z81 an "included" source code file for the assembler
z80asmb.z82 an "included" source code file for the assembler
z80asmb.com an executable file of the assembler
z80asmb.doc this file

8.0 **Expression syntax.**

Expressions follow the syntax diagrams below:

```
 expression
     |
     |
     |----------------------- string -----------------
     |                                          |
      ---------------- arithmetic expression --------------------->>
 arithmetic expression
        |                         --- "|" ---
        |                        |--- "~? ---|
        |                        |--- "-" ---|
        |                        |--- "+" ---|
        |---- "-" ------         |           |
        |---- "+" ------|--- term ------- term ---
         ---- "~" ------         |
                                       --------------------------->>
     term
        |           ----- "&" -----
        |         |----- "\" -----|
        |         |----- "/" -----|
        |         |----- "*" -----|
        |          |             |
         ------------------- factor ------------------------------>>

     factor
        |
        |------------- "'" -- character -- "'" ---------------
        |                                               |
        |------------------------ name ------------------------|
        |                                               |
        |---------------------- number -----------------------|
        |                                               |
        |------------------------ "$" ------------------------|
        |                                               |
         ----- "[" ---- arithmetic expression ---- "]" ----------->>

     number
        |      -------------
        |     |           |
         ------------ digit ------------------ "H" -----
```

```
                                          |----- "B" -----|
                                          |----- "D" -----|
                                           ------------------------->>

        name
          |
          |                   ---- digit ----
          |                 |---- letter ---|
          |                 |             |
           ---- letter --------------------------------------------->>

       string
          |                    ---------------
          |                   |              |
           --- "'" -- character ----- character ----- "'" ---------->>
```

The arithmetic operators are:

monadic operators:
"+" no effect
"-" 2's complementation
"~" 1's complementation
dyadic operators
"+" addition
"-" subtraction
"~" exclusive OR
"|" inclusive OR
"*" multiplication
"/" division
"\" modulus
"&" AND

The multiplicative operators * / \ and & take precedence over the additive operators + - ~ and | but if operators are of equal precedence then evaluation is from left to right in the statement. Expressions within brackets are evaluated first and may be nested to any reasonable degree, the innermost expressions being evaluated first.