Fubukimaru / **asMSX**

<> **Code**   ⊙ Issues 8   ⑁ Pull requests   ▶ Actions   ▥ Projects   📖 Wiki   ⊙ Sec

⑁ master ▾   ···

asMSX / doc / **asmsx.md**

Fubukimaru Added .BIOSVARS to doc ✓   🕘 **History**

⋈ **3 contributors**   👤 👤 👤

Raw   Blame   CL ▢ ✎ 🗑

840 lines (632 sloc) │ 23.9 KB

asMSX - MSX cross assembler

1. Introduction

1.1. Description

asMSX is an assembler for the Z80 processor with a number of extensions to make programming for MSX easy. It is a cross-assembler that runs on Windows, Linux and MacOS. Compile time on a modern PC is near instant comparing to native MSX assemblers. There is no size limit on source file size. Source can be edited in any text editor.

1.2. Features

- supports all official Z80 instructions;
- supports all known unofficial instructions;
- accepts standard Z80 syntax (implicit accumulator);
- works with decimal, hexadecimal, octal and binary numbers;
- supports arithmetic and logic operations in source code;
- supports floating point decimal values by converting them to 16-bit fixed point values;
- math functions: trigonometric, potential etc.
- supports multiple files through inclusion, nesting is allowed;
- supports complete or partial direct inclusion of binary files;
- supports local and global labels;
- supports official MSX BIOS subroutines and system variables using documented names;
- generates binary files loadable from MSX BASIC;
- generates ROM image files;
- supports four MegaROM types: Konami, Konami SCC, ASCII8 and ASCII16;
- generates COM files for MSX-DOS;
- generates CAS files for emulators and WAV files for loading on real MSX computers;
- uses internal Assembler variables;
- generates export symbol table (SYM);
- writes PRINT directive messages to text file (TXT);
- supports conditional assembly;
- integrates with BlueMSX emulator debugger.

1.3. Project goal

asMSX project goal was to create Z80 cross assembler that is flexible, easy to use, reliable and designed from ground up for the development of MSX ROM and MegaROM programs. This goal is fully achieved in current version.

asMSX is a small console program that works very fast. It uses standard libraries present on all PCs.

1.4. Syntax

asMSX implements Z80 assembly language slightly differently from standard Zilog syntax. Major difference is that square brackets `[ ]`, not parentheses `( )` are used for indirect address mode. This design decision allows us to support complex mathematical expressions that use parentheses to order evaluation precedence.

asMSX supports ZILOG directive to retain source level compatibility for existing code. It switches indirect address mode to use parentheses. Mathematical expression evaluation is switched to square brackets as separators.

1.5. Use

asMSX is a command-line program, so the easiest way to use it is from command prompt. To assemble a file use the following command:

```
asmsx [Optional parameters] filename.asm
```

If the extension is not provided, asMSX will assume ".asm" as default. Source code files are expected to be plain 8-bit ASCII text files. asMSX supports text files generated on both Windows and Linux (CR/LF end of line or just LF). On Windows you can assemble a source file by dragging it to asMSX desktop icon. This method is not recommended, since long file name support may not work well. Also please try to avoid dots and other special characters in file names. Dot use for file extension is fine.

asMSX produces messages during assembly.

If no errors are encountered, following files will be generated:

filename.sym: text file with all the symbols defined in program with their decimal values. This file is compatible with BlueMSX debugger. You can use it to make debugging easier. asMSX won't generate SYM file if no constant or variable symbols were defined in program.

filename.txt: text file with messages generated by directives in program code. If PRINT directives are not used, file won't be generated.

Depending on use of output type directives (rom/.rom, .megarom, basic, cas, wav) in your filename.asm, asMSX will generate one or more of the following binary files:

- **filename.z80**: a binary file with no header;
- **filename.bin**: a binary file that can be loaded with `BLOAD"FILENAME.BIN",R` operator in MSX BASIC;
- **filename.com**: an executable file for MSX-DOS;
- **filename.rom**: a binary ROM or MegaROM image file;
- **filename.cas**: a loadable tape image file for MSX emulators. Use `BLOAD"CAS:",R` in MSX BASIC;
- **filename.wav**: a 16 bit 44100Hz (CD quality) mono audio file. It can be loaded on real MSX computer via "tape in" port. Use `BLOAD"CAS:",R` in MSX BASIC.

1.5.1 Command-line parameters

asMSX accepts the following parameters:

- -z: enable standard Z80 Zilog syntax without having .zilog directive in the code.
- -s: silent mode - suppress messages.
- -vv: verbose mode - print more troubleshooting messages.

If you build asMSX from source with YYDEBUG=1, there is one more parameter available:

- -d: print bison debug messages

2. Assembly language

2.1. Assembly syntax

asMSX accepts following line format in source code:

```
[label:]    [directive[parameters]]    [;comments]
[label:]    [opcode[parameters]]       [;comments]
```

Each block in brackets is optional. There is no limit on maximum line length. White space (tabs, spaces and empty lines) are ignored.

Example

```
loop:
        ld a,[0ffffh]    ; reads the master record
```

```
points:
        db 20,30,40,50  ; points table

        org 08000h      ; code offset on page 2
        nop
```

2.2. Labels

Label is a symbolic name for a memory address. Label name should start with either a character or an underscore `_` , the rest could be a character or a number. Label name should always end with a colon `:` . Label name is case sensitive: `LABEL:` and `Label:` are treated as two separate labels.

Example

```
LABEL:
Label:
_alfanumeric:
loop001:
```

Colon defines a label and sets its value equal to memory address of the next opcode or data value. If you want to get address of label, use its name without the colon. You can define local labels by prefixing label name with two "at" symbols `@@` or one "dot". Local label name is only valid up to next global label. This allows to reuse label names for trivial tasks like loops, so you don't have to invent a bunch of different names. It simplifies programming and improves code readability. The following is valid code:

```
Function1:
        ...
@@loop:
        ...
Function2:
        ...
@@loop:
        ...
```

OR

```
Function1:
        ...
.loop:
        ...
Function2:
```

```
        .loop:
              ...
```

2.2.1. Predefined labels

asMSX supports the following predefined labels:

label	value
CHKRAM	0000h
SYNCHR	0008h
RDSLT	000ch
CHRGTR	0010h
WRSLT	0014h
OUTDO	0018h
CALSLT	001ch
DCOMPR	0020h
ENASLT	0024h
GETYPR	0028h
CALLF	0030h
KEYINT	0038h
INITIO	003bh
INIFNK	003eh
DISSCR	0041h
ENASCR	0044h
WRTVDP	0047h
RDVRM	004ah
WRTVRM	004dh
SETRD	0050h
SETWRT	0053h
FILVRM	0056h

100% | Limpar

label	value
LDIRMV	0059h
LDIRVM	005ch
CHGMOD	005fh
CHGCLR	0062h
NMI	0066h
CLRSPR	0069h
INITXT	006ch
INIT32	006fh
INIGRP	0072h
INIMLT	0075h
SETTXT	0078h
SETT32	007bh
SETGRP	007eh
SETMLT	0081h
CALPAT	0084h
CALATR	0087h
GSPSIZ	008ah
GRPPRT	008dh
GICINI	0090h
WRTPSG	0093h
RDPSG	0096h
STRTMS	0099h
CHSNS	009ch
CHGET	009fh
CHPUT	00a2h
LPTOUT	00a5h

Downloads   SetupIperius.exe   100% Limpar

label	value
CNVCHR	00abh
PINLIN	00aeh
INLIN	00b1h
QINLIN	00b4h
BREAKX	00b7h
ISCNTC	00bah
CKCNTC	00bdh
BEEP	00c0h
CLS	00c3h
POSIT	00c6h
FNKSB	00c9h
ERAFNK	00cch
DSPFNK	00cfh
TOTEXT	00d2h
GTSTCK	00d5h
GTTRIG	00d8h
GTPAD	00dbh
GTPDL	00deh
TAPION	00e1h
TAPIN	00e4h
TAPIOF	00e7h
TAPOON	00eah
TAPOUT	00edh
TAPOOF	00f0h
STMOTR	00f3h
LFTQ	00f6h

label	value
RIGHTC	00fch
LEFTC	00ffh
UPC	0102h
TUPC	0105h
DOWNC	0108h
TDOWNC	010bh
SCALXY	010eh
MAPXYC	0111h
FETCHC	0114h
STOREC	0117h
SETATR	011ah
READC	011dh
SETC	0120h
NSETCX	0123h
GTASPC	0126h
PNTINI	0129h
SCANR	012ch
SCANL	012fh
CHGCAP	0132h
CHGSND	0135h
RSLREG	0138h
WSLREG	013bh
RDVDP	013eh
SNSMAT	0141h
PHYDIO	0144h
FORMAT	0147h

label	value
OUTDLP	014dh
GETVCP	0150h
GETVC2	0153h
KILBUF	0156h
CALBAS	0159h
SUBROM	015ch
EXTROM	015fh
CHKSLZ	0162h
CHKNEW	0165h
EOL	0168h
BIGFIL	016bh
NSETRD	016eh
NSTWRT	0171h
NRDVRM	0174h
NWRVRM	0177h
RDBTST	017ah
WRBTST	017dh
CHGCPU	0180h
GETCPU	0183h
PCMPLY	0186h
PCMREC	0189h

2.3. Numeric expression

A numeric expression is a number or a result of operation on numbers.

2.3.1. Numeric formats

There are several popular numeric systems (radices, plural form of radix). Here are a few examples of syntax for such systems:

DECIMAL INTEGER: radix 10 numbers are usually expressed as a group of one or more decimal digits. The only restriction is that you must explicitly express zeroes. This is the numeric system that people use in everyday life.

Example

```
0 10 25 1 255 2048
```

DECIMAL FLOATING POINT: a decimal number with dot separating integer from fraction. Syntax requires dot to be present for the constant to be recognized as a floating point value.

Example

```
3.14 1.25 0.0 32768.0 -0.50 15.2
```

OCTAL: radix 8 numbers could be specified using two conventions. First convention is similar to C, C++ and Java. The number starts with `0` and continues with octal digits `0 .. 7`. The second convention is native to assemblers and is a number with octal digits followed by letter O, lower case or upper case. Second mode is included for compatibility, but is not recommended. Upper case letter `O` is easy to confuse with number zero `0`.

Example

```
01 077 010 1o 77o 10o
```

HEXADECIMAL: radix 16 numbers, very popular in assembly programming. They can be specified using three different conventions.

C, C++ and Java convention is a number that starts with 0x prefix and continues with one or more hexadecimal digits: `0 .. 9`, `a .. f` or `A .. F`.

Second convention is borrowed from Pascal: a group of hexadecimal digits prefixed with $.

The third convention is native to assemblers: a group of hexadecimal digits, followed by letter `h` or `H`. This convention requires first digit to always be numeric. If a hexadecimal number starts with letter, you should prefix the number with '0'.

Example

```
$8a $ff $10
8ah 0ffh 10h
```

BINARY: radix 2 numbers are specified as a group of binary digits `0` and `1`, followed by letter `b` or `B`.

Example

```
1000000b 11110000b 0010101b 1001b
```

2.3.2. Operators

Numeric expressions use operators on numbers in supported numeric systems. Common notation is used for typical arithmetic operators:

- `+` addition;
- `-` subtraction;
- `*` multiplication;
- `/` division;
- `%` modulus (integer remainder from division);

Less common operators borrow from C/C++:

- `<<` left bit shift: shifts specified number of bits left. It is equivalent to a number of multiplications by 2;

- `>>` right bit shift: shifts specified number of bits right. It is equivalent to a number of divisions by 2;

- `|` bit-level OR;

- `&` bit-level AND;

- `^` bit-level XOR;

- `NOT` unary negation;

- `~` logical negation: returns complementing binary number;

- `||` logical OR;

- `&&` logical AND;

- `==` logical equivalent;

- `<` less-than;

- `<=` less-then or equal;

- `>` greater-then;

- `>=` greater-then or equal.

The precedence order is same as in C/C++. Parentheses can be used to explicitly specify parsing precedence in arithmetic expressions.

Example

```
((2*8)/(1+3))<<2
```

Same rules apply to all numbers, including non decimal. Additionally following functions are supported:

- `SIN(X)` sine function, takes input in radians;
- `COS(X)` cosine;
- `TAN(X)` tangent;
- `ACOS(X)` arccosine;
- `ASIN(X)` arcsine;
- `ATAN(X)` arctangent;
- `SQR(X)` square;
- `SQRT(X)` square root;
- `EXP(X)` exponential value of X;
- `POW(X,Y)` returns X raised to the power Y;
- `LOG(X)` logarithm;
- `LN(X)` natural logarithm;
- `ABS(X)` returns the absolute value of X.

`PI` is predefined as a double precision floating point constant. It can be used in numeric expressions.

Example

```
sin(pi*45.0/180.0)
```

It is useful to support non-integer values in Z80 assembler programs by providing simple floating to fixed point conversion mechanism. This is done using two

- `FIX(X)` convert floating point value to fixed point;
- `FIXMUL(X,Y)` multiply two fixed point numbers;
- `FIXDIV(X,Y)` divide X by Y;
- `INT(X)` convert a non-integer value to integer.

`$` is is a special read-only variable. During assembly of program, `$` is replaced with memory address of next opcode or data, during execution it is equal to `PC` register.

2.4. Data definition

You can include data in your assembler program using four different directives:

```
db data,[data...]
defb data,[data...]
dt "text"
deft "text"
```

These instructions define data as 8-bit byte values. dt was included for compatibility with other Z80 assemblers. All four directives are equivalent.

```
dw data,[data...]
defw data,[data...]
```

This will define 16-bit word data.

```
ds X
defs X
```

This will reserve X bytes in memory starting with current memory address.

Special predefined directives are available for reserving memory for variables of conventional sizes, such as byte and word:

- `.byte` reserve one byte of space (8 bits);
- `.word` reserve one word of space, which is equivalent to two bytes (16 bits).

2.5. Directives

Directives are predefined instructions that help control the code and enable additional asMSX features. **Remember, you can use them without first point character.**

`.ZILOG` This directive will switch the use of square brackets and parentheses from the point it is defined on. Parentheses will be used for "memory content at" indirection and brackets to group operations in complex arithmetic expressions.

`.ORG X` This directive is used to force current memory address offset. All subsequent code will be assembled from that offset on.

`.PAGE X` This directive is similar to `.ORG` directive, but instead of bytes, offset is defined in page blocks (16KB). `.PAGE 0` is equivalent to `.ORG 0000h`, `.PAGE 1` is same as `.ORG 4000h`, `.PAGE 2` is same as `.ORG 8000h` and `.PAGE 3` is same as `.ORG 0C000h`.

`.EQU` This directive is used to define constants. Naming rules are mostly the same as for labels. There are no local constants.

`Variable = expression` asMSX can use integer variables. Variables must be initialized by assigning them an integer values. It is possible to perform arithmetic operations with variables.

Example

```
Value1=0
Value1=Value1+1
ld a,Value1
Value1=Value1*2
ld b,Value1
```

`.BIOS` Predefined call address for common BIOS routines, including those specified in the MSX, MSX2, MSX2+ and Turbo-R standards. The usual names are used in upper case. Can be found in asMSX code at the function `msx_bios()` in `dura.y`.

`.BIOSVARS` Predefined system variables. List can be found in Grauw's listing or in asMSX code at the function `msx_bios_vars()` in `dura.y`.

`.ROM` Indicates that a ROM header should be generated. It is important to use .PAGE directive first to define start address. .START directive can be used to indicate the start address for the program.

`.MegaROM [mapper]` Generates header for specified MegaROM mapper. This directive will also set start address to sub-page 0 of selected mapper, so using `ORG`, `PAGE` or `SUBPAGE` directives is not necessary. Supported mapper types are:

- `Konami`: sub-page size is 8 KB, up to 32 sub-pages. Maximum MegaROM size is 256 KB (2 megabits). Page 1 (4000h - 5FFFh) should be mapped to MegaROM sub-page 0 and should not be changed while the program runs.

- `KonamiSCC` : sub-page size is 8 KB, up to 64 sub-pages. Maximum MegaROM size is 512 KB (4 megabits). Supports access to SCC, Konami Sound Custom Chip.

- `ASCII8` : sub-page size is 8 KB, up to 256 sub-pages. Maximum MegaROM size is 2048 KB (16 megabits, 2 megabytes).

- `ASCII16` : sub-page size is 16 KB, up to 256 sub-pages. Maximum MegaROM size is 4096 KB (32 megabits, 4 megabytes).

`.BASIC` Generates the header for a loadable binary MSX-BASIC file. It is important to use .ORG directive to indicate the intended start address for the program. Use .START directive if execution entry point is not at the start of program. The default extension of the output file is BIN.

`.MSXDOS` Produces a COM executable for MSX-DOS. No need for .ORG directive, because COM files are always loaded at 0100h.

`.START X` Indicates the starting execution address for ROM, MegaROM and BIN files, if it is not at the beginning of the file.

`.SEARCH` For ROMs and MegaROMs to that start on page 1 (4000h), it automatically finds and sets slot and subslot on page 2 (8000h). It is equivalent to the following code:

```
call 0138h ;RSLREG
rrca
rrca
and 03h

; Secondary Slot
ld c,a
ld hl,0FCC1h
add a,l
ld l,a
ld a,[hl]
and 80h
or c
ld c,a
inc l
inc l
inc l
inc l
ld a,[hl]

; Define slot ID
and 0ch
or c
```

```
                ; Enable
                call 0024h ;ENASLT
```

`.SUBPAGE n AT X` This macro is used to define different sub-pages in a MegaROM. In MegaROM code generation model of asMSX, all code and data is included in sub-pages, equivalent to logic blocks that mapper operated with. You must provide the number of sub-page and execution entry point.

`.SELECT n AT x` / `.SELECT record AT x` Just like above, this macro selects the sub-page n with execution entry point at x. Specific code that end up being used for this directive depends on selected MegaROM mapper type. It doesn't change the value of any record or affect interrupt mode or status flags.

`.PHASE X` / `.DEPHASE` These two routines enable virtual memory use. Instructions will be assembled to be stored at one memory address, but ready to be executed from another memory address. This is useful for creation of code in ROM image, that is copied to RAM and then executed. The effect is that label values will be calculated for supplied address. .DEPHASE reverts assembler behaviour to normal, although ORG, PAGE or SUBPAGE will have the same effect.

`.CALLBIOS X` Calls a BIOS routine from MSX-DOS program. It is equivalent to following code:

```
        LD IY,[EXPTBL-1]
        LD IX,ROUTINE
        CALL CALSLT
```

`.CALLDOS X` Calls MSX-DOS function. It is equivalent to following code:

```
        LD C,CODE
        CALL 0005h
```

`.SIZE X` Sets the output file size in Kilobytes.

`.INCLUDE "file"` Includes source code from an external file.

`.INCBIN "file" [SKIP X] [SIZE Y]` Injects the contents of a binary file into program. Optional SIZE and SKIP parameters allow to include a number of bytes, starting at specified offset.

`.RANDOM(n)` Generates a random integer number from the range of 0 to n-1. Provides better entropy than the Z80 R register.

.DEBUG "text" Adds text to assembled program that is visible during debugging, but does not affect the execution. BlueMSX debugger supports this extended functionality.

.BREAK [X] / .BREAKPOINT [X] Defines a breakpoint for BlueMSX debugger. It doesn't affect the execution, but it should be removed from the final build. If the direction is not indicated, the breakpoint will be executed in the position in which it has been defined.

REPT n / ENDR This macro allows you to repeat a block given number of times. Nesting allows generation of complex tables. There is a restriction: the number of repetitions must be defined as integer number, it can't be calculated from a numeric expression.

Example

```
REPT 16
  OUTI
ENDR

X=0
Y=0
REPT 10
  REPT 10
    DB X*Y
    X=X+1
  ENDR
  Y=Y+1
ENDR
```

.PRINTTEXT "text" / .PRINTSTRING "text" Prints a text in the output text file.

.PRINT expression / .PRINTDEC Prints a numeric expression in decimal format.

.PRINTHEX expression Prints a numeric expression in hexadecimal format.

.PRINTFIX expression Prints a fixed-point numeric value.

.CAS ["text"] / .CASSETTE ["text"] Generates a tape file name in output file. This only works if output file type is loadable BASIC program, ROM in page 2 or Z80 binary blob without header. Supplied name can be used to load the program from the tape, maximum length is 6 characters. If not explicitly specified, it is set to name of output file.

.WAV ["text"] Like the previous command, but instead of CAS file, it generates a WAV audio file that can be loaded directly on a real MSX through tape-in port.

`.FILENAME ["text"]` Using this directive will set the name of the output file. If not explicitly specified, source file name will be used, plus appropriate extension.

`.SINCLAIR` **Currently broken** This directive sets the output file type to TAP format with appropriate header. It is intended for loading on ZX Spectrum emulators or real hardware if you have a working playback application.

2.6. Comments

It is highly recommended to use comments throughout the assembler source code. asMSX supports comments in a variety of formats:

```
; Comment
```

Single line comment. This is the standard for assembler listings. Entire line up to carriage return is ignored.

```
// Comment
```

Same as above. Two consecutive slashes is a convention from C++ and C since C99.

```
/* Comments */
{ Comments }
```

C/C++ and Pascal style multi line comments. All text between the delimiters is skipped during assembly.

```
-- Comment
```

Ada-style single line comment.

2.7. Conditional assembly

asMSX includes preliminary support for conditional assembly. The format of a conditional assembly block is as follows:

```
IF condition
   Instruction
ELSE
   Instruction
ENDIF
```

The condition can be of any type, consistent with C/C++ rules. A condition is considered true if evaluation result is non-zero.

If condition is true, asMSX will assemble code that follows `IF`. If the condition is false, code after `ELSE` will be assembled.

`ELSE` block is optional for `IF`.

`ENDIF` is mandatory, it closes conditional block.

Current IF nesting limit is 15. It may become unlimited in future rewrite.

Example

```
IF (computer==MSX)
  call MSX1_Init
ELSE
  IF (computer==MSX2)
    call MSX2_Init
  ELSE
    IF (computer==MSX2plus)
      call MSX2plus_Init
    ELSE
      call TurboR_Init
    ENDIF
  ENDIF
ENDIF
```

In addition, all code, directives and macros will be executed according to conditions, this enables creation of the following structures:

```
CARTRIDGE=1
BINARY=2
format=CARTRIDGE
IF (format==CARTRIDGE)
  .page 2
  .ROM
ELSE
  .org 8800h
  .Basic
ENDIF
.START Init
```

There is a limitation on conditional instructions: `IF` condition, `ELSE` and `ENDIF` must appear on their own separate lines. They can't be mixed with any other instructions or macros. The following code will fail with current asMSX:

```
IF (variable) nop ELSE inc a ENDIF
```

It should be written as follows:

```
IF (variable)
  nop
ELSE
  inc a
ENDIF
```

`IFDEF` condition could be used to branch code assembly using a defined symbol as argument.

Example

```
IFDEF tests
  .WAV "Test"
ENDIF
```

This snippet will generate a WAV file only if the label or variable tests was previously defined in the source code.

BEWARE!

- `IFDEF` **will only recognize a label if it is defined before** `IFDEF` **.**
- **Don't use** `INCLUDE` **inside an** `IFDEF` **or an** `IF` **. It doesn't work.**