ASMotor v0.1.0

Carsten Elton Sørensen

November 13, 2010

ASMotor

Introduction

ASMotor is a portable and generic assembler engine and development system written in ANSI C and licensed under the GNU Public License v3. The package consists of the assembler, the librarian and the linker. It can be used as either a cross or native development system.

ASMotor first saw the light of day as RGBDS, a Gameboy development system. RGBDS used flex and bison and used two pass assembling. Since then it has been rewritten and now features a custom lexer and parser and the assembler does its work in only one pass, all of which make it much faster than the first versions.

ASMotor is largely compatible with sources written for RGBDS although there have been changes that will break some of this compatibility. The bump in the major version number indicates that compatibility is not ensured.

Features

CPU architectures:

- The Gameboy Z80 derivative
- MC680x0
- 6502
- MIPS32

Object formats:

- xobj (ASMotor generic format)
- binary
- Amiga hunk object
- Amiga executable

The assembler asmotor

Invoking asmotor

Depending on the target CPU, the executable to invoke will be named motor, followed by the CPU architecture name. motorgb (for Gameboy), motor68k (for MC680x0), motor6502 (6502 and derivatives) or motormips.

Listing 1: Supported command line options

```
-b<AS>  Change the two characters used for binary constants
        (default is 01)
-e(l|b) Change endianness
-f<f>   Output format, one of
        x - xobj (default)
        b - binary file
        g - Amiga executable file
        h - Amiga object file
-i<dir> Extra include path (can appear more than once)
-o<f>   Write assembly output to <file>
-v      Verbose text output
-w<d>   Disable warning <d> (four digits)
-z<XX>  Set the byte value (hex format) used for uninitialised
        data (default is ? for random)
```

Syntax

The syntax is line based. A valid line consists of an optional label, an optional machine or assembler instruction and an optional comment. Labels are described on the following page, assembler instructions on page 12.

Comments

Comments are ignored during assembling. They are an important part of writing code, this is especially true for assembly where comments are essential for documenting what a function does as it's not immediatly obvious as it may be in a high level language.

Comments are usually started with a semi-colon and end at the end of the line. A comment may also be started with a whitespace character (including a line break) followed by an asterisk.

Listing 2: Comment examples

```
* These are comment examples
Label:  moveq   #1,d0   ;load register d0 with the value 1
        move.l  d0,d1   *copy it to d1
```

Labels

One of the assembler's main tasks is to keep track of addresses so you don't have to remember obscure numbers but can use a meaningful name instead, a label. Labels are always placed at the beginning of a line.

Labels end with zero, one or two colons. If the label ends with two colons it will be automatically exported[1]

Symbols and labels are always case-sensitive.

Global labels

Global labels start with a character from A to Z (or their lower case equivalents) or an underscore. After that the characters a-z, A-Z, 0-9, _, @ and # may be used. A global label also marks the beginning of a new scope for local labels.

Listing 3: Label examples

```
GlobalLabel
AnotherGlobalLabel:
ExportedLabel::
```

Local labels

The assembler supports local labels. Local labels start with the . character, followed by a character in the range a-z, A-Z or an underscore, after which the characters a-z, A-Z, 0-9, _, @ and # may be used. A local label is considered local to the scope in which it is defined, a scope begins with a global label and ends with the next global label. Local labels can only be referenced within the scope they are defined.

Listing 4: Local label examples

```
GlobalLabel:
.locallabel
AnotherGlobalLabel:
```

[1] see XDEF reference on the following page and examples on the next page

Exporting and importing labels

Most of the time programs consists of several source files that are assembled individually and the resulting object files then linked into an executable. This improves the time spent assembling and help manage a project.

To export a symbol (to let other source files use the symbol,) you use the keyword EXPORT (or its synonym XDEF) followed by the symbol that should be exported. To import a symbol (to make an externally defined symbol available in the current source file,) the keyword IMPORT (or its synonym XREF) is used.

Instead of using EXPORT or XDEF to export a label, the label may end with two colons.

Often you will want to make a header file for these definitions that other source files can include for easy access to the symbols you have exported. However, if you want to include the file yourself, for instance if it contains structure definitions, it becomes slightly complicated - if a symbol is first imported in the include file and later appears as a label, you have multiple definitions of the symbol.

The assembler provides a third keyword, GLOBAL, that either imports or exports a symbol, depending on whether it appears as a label in the current source file.

Listing 5: Import and export examples

```
EXPORT  GlobalLabel
XREF    ImportedLabel
GLOBAL  AnotherImport,AnExportedLabel
GlobalLabel:
AnExportedLabel:
AutomaticExport::
```

Note that all three assembler instructions accept a comma separated list of labels.

Integer symbols

Instead of hardcoding constants it's often better to give them a name. The assembler supports two kinds of integer symbols, one that is constant and one that may change its value during assembling. The assembler instruction EQU is used to define constants and SET is used for variables. Instead of SET it's also possible use =. Note that an integer symbol is never followed by a colon.

Listing 6: Constant and variable examples

```
INTF_MASTER EQU $4000
MyCounter   SET 0
MyCounter   =   MyCounter+1 ;Increment MyCounter
```

Integer symbols are often used to define the offsets of structure members. While the EQU instruction can be used for this it quickly becomes cumbersome when adding, reordering or removing members from the structure. The assembler provides a group of instructions to make this easier, the RS group of instructions.

Command	Meaning
RSRESET	Resets the __RS counter to zero
RSSET constexpr	Sets the __RS counter to the value of constexpr
Symbol: RB constexpr	Sets Symbol to __RS and adds constexpr to __RS
Symbol: RW constexpr	Sets Symbol to __RS and adds 2*constexpr to __RS
Symbol: RL constexpr	Sets Symbol to __RS and adds 4*constexpr to __RS

Listing 7: RS example

```
            RSRESET
str_pStuff  RW 1
str_tData   RB 256
str_bCount  RB 1
str_SIZEOF  RB 0
```

The example defines four equated symbols and their values:

Symbol	Value
str_pStuff	0
str_tData	2
str_bCount	258
str_SIZEOF	259

Like labels constants can also be exported, if the chosen object format supports it.

String symbols

String symbols are used to assign a name to an often used string. These symbols are expanded to their value whenever the assembler encounters the assigned name.

Listing 8: String symbol example

```
COUNTREG EQUS "[hl+]"
         ld   a,COUNTREG
```

The example above will be interpreted as

Listing 9: String symbol expansion

```
         ld   a,[hl+]
```

String symbols can also be used to define small macros

Listing 10: Multiline string symbol

```
PUSHA    EQUS "push af\npush bc\npush de\npush hl\n"
```

Predeclared symbols

The assembler declares several symbols:

Name	Contents	Type
@, *	Current PC value	EQU
__RS	__RS counter	SET
__NARG	Number of arguments passed to macro	EQU
__LINE	The current line number	EQU
__FILE	The current filename	EQUS
__DATE	Todays date	EQUS
__TIME	The current time	EQUS

Code, data and variables

From the assembler's point of view, there is no difference between code and data. Code is entered by using the mnemonics described in the relevant backend chapters.

Listing 11: Code example

```
mfhi t0
nop
nop
mult t0,t1
```

Data (or instructions) can also be entered using data declaration statements:

```
DC.W $4E75
DC.B "This is a string",0
```

The data declaration statements may be called something different depending on the CPU backend. Please refer to the relevant backend chapter.

Data can also be declared by simply including a binary file directly from the file system:

Listing 12: Binary file inclusion

```
        SECTION "Graphics",DATA
Ship:   INCBIN "Spaceship.bin"
```

Sections

Code, data and variables are organised in sections. Before any mnemonics or data declarations can be used, a section must be declared.

Listing 13: A simple code section

```
SECTION "A_Code_Section",CODE
```

This will switch to the section "A_Code_Section" if it is already known (and its type matches), or declare it if it doesn't. DATA may also be used instead of CODE, they are synonymous.

Variables are usually placed in a BSS section. BSS section cannot contain initialised data, typically only the DS command is used. However, it is also possible to use the DB, DW and DL commands without any arguments.

Listing 14: Variable section

```
        SECTION "Variables",BSS
Foo:    DB              ; Reserve one byte for Foo
Bar:    DW              ; Reserve a word for Bar
Baz:    DS str_SIZEOF   ; Reserve str_SIZEOF bytes for Baz
```

If the chosen object output format supports it, you can force a section into a specific address:

Listing 15: Section fixed to address

```
        SECTION "LoadSection",CODE[$F000]
Code:   xor a
```

The different CPU backends may support additional section types and other options. Please refer to the relevant chapters.

The section stack

A section stack is available, which is particularly useful when defining sections in included files (or macros) and it's necessary to preserve the section context for the program that included the file or called the macro.

POPS and PUSHS provide the interface to the section stack. PUSHS will push the current section context on the section stack. POPS can then later be used to restore it.

The macro language

The macro language is an interpreted language that offers features which make an assembly source more readable. It also offers fine control over how the source is assembled.

Integer expressions

Whenever the assembler expects an integer, an integer expression may be used. Integer expressions are always evaluated using signed 32 bit math.

The simplest integer expression is a number.

An expression is said to be constant when it doesn't change its value during linking. This basically means that you can't use labels in those expressions. The instructions in the macro-language all require expressions that are constant

Integer literals

The assembler supports several numeric formats:

Hexadecimal $0123456789ABCDEF (case-insensitive)

Decimal 0123456789

Binary %01

Fixedpoint (16.16) 01234.56789

Character "ABYZ"

Operators

Several operators can be used to build up integer expressions. In order of precedence they are:

Operator	Meaning
( )	Precedence override
FunctionName(...)	Function call
~ + -	Unary bitwise not, plus, negation
* / ~/ ** //	Multiply, divide, modulo, fixedpoint multiply and divide
<< >>	Shift left, shift right
& ^ |	Bitwise and, bitwise exclusive or, bitwise or
+ -	Add, subtract
~= == <= >= < >	Comparison operators: not equal, equal, less than or equal, greater than or equal, less than, greater than
&& ||	Boolean and, boolean or
~!	Unary boolean not

The result of the boolean operators and comparison operators is zero if when *false* and non-zero when *true*.

Fixed point math

The assembler supports fixed point constants, which are normal 32 bit constants where the upper 16 bits are used for the integer part and the lower 16 bits are used for the fraction (65536ths).

Fixed point values can be used in normal integer expressions, some integer operators like plus and minus work the same whether the operands are integer or fixed point. A fixed point number can be converted to an integer by shifting it right 16 bits, as this will discard the fractional part and leave the integer part at the right bit position. It follows that you can convert an integer to a fixed point number by shifting it left 16 positions.

Other fixed point operations require more precision than 32 bit math provides, the following fixed point functions are therefore available:

Name	Operation
x // y	x/y
x ** y	$x * y$
SIN(x)	$sin(x)$
COS(x)	$cos(x)$
TAN(x)	$tan(x)$
ASIN(x)	$sin^{-1}(x)$
ACOS(x)	$cos^{-1}(x)$
ATAN(x)	$tan^{-1}(x)$
ATAN2(x,y)	Angle of the vector $\begin{bmatrix} x \\ y \end{bmatrix}$

A circle has 1.0 fixed point degrees (65536 integer), sine values are in the range $[-1.0; 1.0]$

These functions are particularly useful for generating various tables:

Listing 16: Generate a 256 byte sine table with values between 0 and 128

```
ANGLE   SET     0.0
        REPT    256
        DB      (64.0**SIN(ANGLE)+64.0)>>16
ANGLE   SET     ANGLE+1.0/256
        ENDR
```

String expressions

Whenever the assembler excepts a string literal, a string expression may be used instead. The simplest string expression is a string literal - a string contained in double quotes. A string literal can also contain special characters, such as newlines and tabs by using escape sequences initiated by a backslash:

Sequence	Meaning
\\	\
\"	“
\{	{
\}	}
\n	Newline ($0A)
\t	Tab ($09)
\0 .. \9	Value of macro argument
\@	Unique label suffix

The macro arguments are only valid within a macro, the \@ sequence is valid in macros and REPT blocks.

Within a string literal it's possible to embed the value of a symbol. This is done by encloding the symbol name in curly brackets:

Listing 17: Symbol embedded in string literal

```
StringSymbol EQUS "A String"
              DB   "Store the value {StringSymbol}",0
```

As a shorthand, a symbol can simply be surrounded by curly brackets outside a string literal, to convert its value to a string expression:

Listing 18: Converting a symbol to string expression

```
StringSymbol EQUS "Another string"
              DB   {StringSymbol} ; The value of StringSymbol will be stored
```

String functions and properties

Several functions that work on string expressions are available. Some of these return strings and some return integers. Functions that return an integer can be used as part of integer expressions, when a string is returned the function can be used in a string expression.

Name	Type	Result
s.length	integer	The number of characters in *s*
s1.compareto(*s2*)	integer	Negative if *s1* is alphabetically < *s2*, 0 if equal, positive if >
s1.indexof(*s2*)	integer	The position of *s2* within *s1*, -1 if not found
s1==*s2*	integer	Non-zero if *s1* is equal to *s2*, otherwise zero
~= < <= > >=	integer	String comparison operators.
s1+*s2*	string	String concatenation, *s1* followed by *s2*
s.slice(*pos*,*count*)	string	*count* characters from *s*, starting at *pos*
s.toupper()	string	Upper case version of *s*
s.tolower()	string	Lower case version of *s*

The *pos* parameter for .slice() may also be a negative number, in which case the position is relative to the end of the string, with -1 being the last character of the string. *count* may be completely omitted, in which case characters from *pos* until the end of the string is returned.

Outputting messages

While assembling it is possible to cause the assembler to print out various user defined messages.

Simple diagnostic messages are possible with PRINTT, PRINTF and PRINTV:

```
PRINTT "A simple message\n" ; Remember \n for newline
PRINTV (2+3)/5              ; Prints an integer
PRINTF 3.14**,2             ; Prints a fixed point value
```

In macros it can be helpful to warn the user of a wrong argument or completely abort the assembly process. This is possible with the FAIL and WARN

commands. FAIL and WARN take a string as the only argument and will print it out as a regular warning or error with a line number.

FAIL stops assembling immediately while WARN continues after printing the errormessage.

Listing 19: FAIL and WARN example

```
IF (\1)<42
WARN "Argument should be >= 42"
ENDC

IF (\1)>100
FAIL "Argument MUST be <= 100"
ENDC
```

Including files

The INCLUDE command is used to process another assembly file and then return to the current file when done. If the file is not found in the current directory, the include path list will be searched. INCLUDE files may be nested.

Listing 20: INCLUDE example

```
INCLUDE "irq.inc"
```

Repeating blocks

To repeat a block it can be placed inside a REPT/ENDR structure. The REPT construct repeats the block a specified number of times.

Listing 21: Basic REPT example

```
REPT 4
add  a,c
ENDR
```

REPT can also be used to repeat macro language constructs such as IF/ENDC and used to construct various tables (example on page 14.)

Conditional assembling

The IF/ELSE/ENDC commands are used to conditionally include or exclude parts of an assembly file.

```
IF     2+2==4
PRINTT "2+2==4\n"
ELSE
PRINTT "2+2!=4\n"
ENDC
```

The ELSE block is optional.

While the integer operators can be used to test for many conditions, the assembler also supports the traditional IF commands:

Command	True when
IFC *s1,s2*	The string *s1* equals the string *s2*
IFNC *s1,s2*	The string *s1* is different from the string *s2*
IFD *symbol*	The symbol *symbol* is defined
IFND *symbol*	The symbol *symbol* is not defined
IFEQ *n*	*n* equals zero
IFNE *n*	*n* is not equal to zero
IFGE *n*	*n* is greater than or equal to zero
IFGT *n*	*n* is greater than zero
IFLE *n*	*n* is less than or equal to zero
IFLT *n*	*n* is less than zero

Macros

One of the most useful features of an assembler is the ability to write macros. Macros also provide a method of passing arguments to them and they can then react to the input using conditional assembling constructs.

Listing 22: Macro example

```
MyMacro: MACRO
         ld   a,80
         call MyFunc
         ENDM
```

The example above is a very simple macro. You use the macro by using its name where you would normally use an instruction.

Listing 23: Macro expansion

```
         add a,b
         ld  sp,hl
         MyMacro  ;This will be expanded
         sub a,87
```

When the assembler sees MyMacro it will insert the macro definition, the text enclosed in MACRO/ENDM.

Macro loops

Often macros will contains loops, such as:

Listing 24: Macro with loop

```
LoopMacro: MACRO
           xor a,a
.loop      ld  [hl+],a
           dec c
           jr  nz,.loop
           ENDM
```

This will work fine, until you start using the macro more than once per scope. To get around this problem there is a special label string equate called \@ that you can append to your labels and it will then expand to a unique string. \@ also works in REPT-blocks.

Listing 25: Macro with unique local label

```
LoopMacro: MACRO
           xor a,a
.loop\@    ld  [hl+],a
           dec c
           jr  nz,.loop\@
           ENDM
```

Arguments

LoopMacro above could be improved, it would be better if the user didn't have to preload the registers with values and then call the macro. Fortunately it's possible to pass arguments to a macro, the LoopMacro example would then be able to load the registers itself.

In macros you can get the arguments by using the special macro string equates \1 through \9, \1 being the first argument specified on the call of the macro.

Listing 26: Macro with two arguments

```
LoopMacro: MACRO
           ld  hl,\1
           ld  c,\2
           xor a,a
.loop\@    ld  [hl+],a
           dec c
           jr  nz,.loop\@
           ENDM
```

The macro now accepts two arguments. The first being an address and the second being a byte count. The macro will then set all bytes in this range to zero.

Listing 27: Invoking Macro with arguments

```
LoopMacro MyVars,54
```

You can specify up to nine arguments when calling a macro. Arguments are passed as string equates, there's no need to enclose them in quotes. As the arguments are considered strings, this means that it's a good idea to use brackets around \1-\9 if you perform further calculations on them. Consider the following scenario:

Listing 28: Macro argument caveat

```
PrintValue: MACRO
            PRINTV \1*2
            ENDM

            PrintValue 1+2
```

Here the assembler will print the value 5 on screen and not 6 as you might expect. The solution is to enclose \1 in brackets:

Listing 29: Macro argument caveat fix

```
PrintValue: MACRO
            PRINTV (\1)*2
            ENDM
```

Sometimes it may be necessary to pass a comma into a macro. To do this, the macro argument can be enclosed in angle brackets:

Listing 30: Passing a comma to a macro

```
PrintString: MACRO
             lea   .string\@(PC),a0
             jsr   _print
             bra   .skip\@
.string\@    dc.b \1
             EVEN
.skip\@
             ENDM

             PrintString <"Hello,"," world",0>
```

The special argument \0

Particularly useful on MC680x0, it's also possible to use the special argument \0. To pass a value into \0 you append a dot (.) followed by the value to the macro name.

Listing 31: Macro argument \0

```
push: MACRO
      movem.\0 \1,-(sp)
      ENDM

      push.l   d0-a6
```

SHIFT

SHIFT is a command only available in macros and particularly useful in REPT-blocks. It "shifts" the macro arguments one position "to the left" - \1 will get \2's value, \2 will get \3's value and so forth.

Final notes

A colon (:) following the macro-name is required, macros cannot be exported or imported and it's valid to call a macro from a macro (even the same one for recursive behaviour).

Gameboy backend

Numeric formats

The Gameboy backend supports an additional numeric format:

Gameboy graphics ‘00112233

The values are actually pixel values. The values are converted from chunky data format to the planar format as used in the Gameboy.

Sections

The Gameboy supports several banks of code and data, in addition to fixing a section to a specific address, it's also possible to force it into a specific bank:

Listing 32: Section fixed to address and bank

```
SECTION "FixedSection",DATA[$1100],BANK[3]
```

It's also possible to only specify the bank:

Listing 33: Section fixed to bank

```
SECTION "FixedSection",CODE,BANK[3]
```

Index