NAME

asm6809—6809 cross-assembler

SYNOPSIS

asm6809 [*OPTION*]… [*SOURCE-FILE*]…

DESCRIPTION

asm6809 is a portable macro cross assembler targeting the Motorola 6809 and Hitachi 6309 processors. These processors are most commonly encountered in the Dragon and Tandy Colour Computer.

OPTIONS

-B, **--bin**
: output raw binary file (default)

-D, **--dragondos**
: output DragonDOS binary file

-C, **--coco**
: output CoCo RS-DOS (“DECB”) segmented binary file

-S, **--srec**
: output Motorola SREC file

-H, **--hex**
: output Intel hex record file

-e, **--exec** *addr*
: EXEC address (for output formats that support one)

-8, **-9**, **--6809**
: use 6809 ISA (default)

-3, **--6309**
: use 6309 ISA (6809 with extensions)

-d, **--define** *sym*[=*number*]
: define a symbol

--setdp *value*
: initial value assumed for DP [undefined]

-o, **--output** *file*
: output filename

-l, **--listing** *file*
: create listing file

-s, **--symbols** *file*
: create symbol table

-q, **--quiet**
: don't warn about illegal (but working) code

-v, **--verbose**
: warn about explicitly inefficient code

--help
: show help

--version
: show program version

If more than one *SOURCE-FILE* is specified, they are assembled as though they were all in one file.

USAGE

Text is read in and parsed, then as many passes are made over the parsed source as necessary (up to a limit), until symbols are resolved and addresses are stable. The fastest or smallest representation should always be chosen where there is ambiguity.

Output formats are: Raw binary, DragonDOS binary, CoCo RS-DOS (“DECB”) binary, Motorola SREC, Intel HEX.

Home page: <http://www.6809.org.uk/asm6809/>

Differences to other assemblers

Motorola syntax allows a comment to follow any operands, separated from them only by whitespace. To an extent, this assembler accepts that, but be aware that as spaces are allowed within expressions, if the comment looks like it is continuing an expression it will generate bad code (or raise an error if the result is syntactically incorrect). Example:

```
0000  8605                lda     #5
0002  C60A                ldb     #5 * 2  twice first number
```

A strict Motorola assembler would generate bytes C6 05 for the second line, as the "* 2" would be ignored. For consistency, it is best to introduce end of line comments with a **;** character. An asterisk (*****) can introduce whole line comments.

An unquoted semicolon always introduces a comment. The alternate form of the 6309 instructions **AIM**, **OIM**, etc. listed in some documentation that uses a semicolon as a separator is not accepted.

A symbol may be forward referenced; any time a reference is unresolvable, another pass is triggered, up to some defined maximum.

In 6809 indexed addressing, the offset size will default to the fastest possible form, e.g. if the offset is an expression that happens to evaluate to zero, the *no offset* form will be used. Prepend **<<** to coerce a 5 bit offset, **<** to coerce 8 bits or **>** to coerce 16 bits.

asm6809 currently has no support for OS-9 modules or multiple object linking.

Program syntax

Program files are considered line by line. Each line contains up to three fields, separated by whitespace: label, instruction and arguments. An unquoted semicolon (**;**) indicates that the rest of the line is to be considered a comment. Whole line comments may be introduced with an asterisk (*****). Motorola-style end of line comments without a **;** are accepted, but see the notes about assembler differences.

Any label must appear at the very beginning of the line. If a label is omitted, whitespace must appear before the operator field. Certain pseudo-ops may affect a label's meaning, but usually labels define a symbol referring to the current position in the code (Program Counter, or PC).

The instruction field contains either an instruction op-code (mnemonic), a pseudo-op (assembler directive), or a macro name for expansion.

Pseudo-ops allow conditional assembly and inline data, can affect code placement and symbol values and be used to include further files inline. See the section on Pseudo-ops for more information.

Arguments are a comma-separated list: either instruction operands or arguments to a pseudo-op or macro. Permitted arguments are specific to the instruction or pseudo-op, but in general they may be:

- An expression.
- A register name, with optional pre-decrement or post-increment.
- A nested list surrounded by **[** and **]**. This is generally only used to indicate indirect indexed addressing.

In addition, any argument may be preceded by:

- **#**, indicate immediate value.
- **<<**, force 5-bit index offset.
- **<**, force direct addressing, 8-bit value or 8-bit index offset.
- **>**, force extended addressing, 16-bit value or 16-bit index offset.

Expressions

Expressions are formed of:

- A decimal number.
- An octal number preceded by **@** or with a leading **0**.

- A binary number preceded by **%** or **0b**.
- A hexadecimal number preceded by **$** or **0x**.
- A floating point number: decimal digits surrounding exactly one full stop (**.**).
- A single quote followed by any ASCII character (yielding the ASCII value of that character).
- A symbol name, local forward reference or local back reference.
- Any of the above prefixed with a unary minus (**−**) or plus (**+**).
- A string delimited either by double quotes or **/**.
- A combination of any of the above with arithmetic, bitwise, logical or relational operators.
- Parenthesis to specify precedence.

The assembler uses multiple passes to resolve expressions. If an expression refers to a symbol that cannot currently be resolved, an extra pass is triggered. Similarly, if a symbol is assigned a value (e.g. by an **EQU** pseudo-op) that differs to its value on the previous pass, another is triggered until it becomes stable.

When not directly used for their contents (e.g. by **FCC**), strings can be used in place of integer values. The ASCII value of each character is used to represent 8 bits of the integer result up to 32 bits. Example:

```
0000  CC443A                ldd     #"D:"
```

Operators

The following operators are available, listed in descending order of precedence (where operators share a precedence, left-to-right evaluation is performed):

Operator	Description
+	unary plus
−	unary minus
! ~	logical, bitwise NOT
*****	multiplication
/	division
%	modulo
+	addition
−	subtraction
<<	bitwise shift left
>>	bitwise shift right
< <=	relational operators
> >=	relational operators
==	relational equal
!=	relational not equal
&	bitwise AND
^	bitwise XOR
|	bitwise OR
&&	logical AND
||	logical OR
?:	ternary operator

Division always returns a floating point result. Other arithmetic operators return integers if both operands are integers, otherwise floating point. Bitwise operators and modulo all cast their operands to integers and return an integer. Relational and logical operators result in 0 if false, 1 if true. Integer calculations are performed using the platform's *int64_t* type, floating point uses *double*.

Conditional assembly

The pseudo-ops **IF**, **ELSIF**, **ELSE** and **ENDIF** guide conditional assembly. **IF** and **ELSIF** take one argument, which is evaluated as an integer. If the result is non-zero, the following code will be assembled, else

it will be skipped. Undefined symbols encountered while evaluating the condition are interpreted as zero (false) rather than raising an error.

Conditional assembly pseudo-ops are permitted within macro definitions and will be evaluated at the time of expansion, therefore positional variables can be used to affect macro expansion.

Sections

Code can be placed into named sections with the **SECTION** pseudo-op. This can make breaking source into multiple input files more comfortable. Without **ORG** or **PUT** directives, sections will follow each other in memory in the order they are first defined.

Within each section, there may exist multiple spans of discontiguous data. Certain output formats are able to represent this, for the others (e.g. DragonDOS), the spans are combined first, with the gaps between them padded with zero bytes.

Local labels

Local labels are considered local to the current *section*. A local label is any decimal number used in the label field, and the same local label may appear mulitple times, unlike other labels.

As an operand, a decimal number followed by **B** or **F** is considered to be a back or forward reference to the previous or next occurrence of that numerical local label in the section.

In this example, the **1** label occurs twice, but each use of **1B** refers to the closest one searching backwards:

```
0000  8E0400    scroll      ldx     #$0400
0003  EC8820    1           ldd     32,x
0006  ED81                  std     ,x++
0008  8C05E0                cmpx    #$05e0
000B  25F6                  blo     1B
000D  CC6060                ldd     #$6060
0010  ED81      1           std     ,x++
0012  8C0600                cmpx    #$0600
0015  25F9                  blo     1B
0017  39                    rts
```

An exclamation mark (**!**) in the label field is treated as a local label numbered zero. Operands of **<** and **>** are considered equivalent to **0B** and **0F** respectively, and can therefore refer to the **!** local label. This is included for compatibility with other assemblers.

As local labels can be repeated, their position is used to distinguish them. For this reason, all file inclusions and macro expansion must occur during the first pass so that the absolute line count at which each local label is encountered remains the same between passes.

Macros

Start a macro definition by specifying a name for it in the label field, and **MACRO** in the instruction field. Finish the definition with **ENDM** in the instruction field.

Use a macro by specifying its name in the instruction field. Any arguments given will be available during expansion as a positional variable.

Positional variables can be used within strings, or pasted to form symbol names. In either case, they must be quoted or they will be passed by value, which will result in an error if they do not correspond to valid symbols by themselves.

The positional variables are referred to with **\{1}**, **\{2}**, … , **\{***n***}**. For the first nine arguments, the braces are not required, so **\1**, **\2**, … , **\9** are valid alternatives. For compatibility with the TSC Flex assembler, another form is accepted: **&{1}**, **&{2}**, … , **&{***n***}**. Within a string, the shorter **&1**, **&2**, … , **&9** is still valid, but as this can be confused with bitwise AND, it is not permitted elsewhere.

Here's a silly example demonstrating positional variables and symbol pasting. Consider the following macro definition and utilising code:

```
go_left         equ     -1
go_right        equ     +1
move            macro
                lda     x_position
                adda    #go_\1
                sta     x_position
                endm
do_move
                move    "right"
                rts
x_position      rmb     1
```

The main code generated is as follows:

```
0000            do_move
0000                        move    "right"
0000  B60009                lda     x_position
0003  8B01                  adda    #go_\1
0005  B70009                sta     x_position
0008  39                    rts
```

Pseudo-ops

Conditional assembly:

IF *condition*
: Subsequent lines are assembled only if *condition* evaluates to true (non-zero).

ELSIF *condition*
: Subsequent lines are assembled only if all preceding **IF** and **ELSIF** pseudo-ops evaluated to false (zero) and *condition* evaluates to true (non-zero).

ELSE Subsequent lines are assembled only if all preceding **IF** and **ELSIF** pseudo-ops evaluated to false (zero).

ENDIF Terminate an **IF** statement.

Macro definition:

MACRO Start defining a macro. The macro's name shall be in the label field. Subsequent lines up to the enclosing **ENDM** pseudo-op will not be assembled until the macro is expanded. Macro definitions may be nested; that is, using a macro may define another macro.

ENDM Finish a macro definition started with **MACRO**.

Inline data:

FCB *value*[,*value*]…
FCC *value*[,*value*]…
: Form Constant Byte. Each *value* is evaluated either to a number or a string. Numbers are truncated to 8 bits and stored directly as bytes. For strings, the ASCII value of each character is stored in sequential bytes.

  Historically, **FCB** handled bytes and **FCC** (Form Constant Character string) handled strings. **asm6809** treats them as synonymous, but is rather more strict about what is allowed as a string delimiter.

FCN *value*[,*value*]…
: Identical to **FCC**, but a terminating zero byte is stored after the data. Included to increase compatibility with other assemblers.

FDB *value*[,*value*]…
: Form Double Byte. Each *value* is evaluated to a number, which is truncated to 16 bits and stored as two successive bytes (big-endian).

FQB *value*[,*value*]…
: Form Quad Byte. Each *value* is evaluated to a number, which is truncated to 32 bits and stored as four successive bytes (big-endian).

FILL *value*,*count*
: Insert *count* bytes of *value*. This is effectively the same as the two-argument form of **RZB** with its arguments swapped.

RZB *count*[,*value*]
ZMB *count*[,*value*]
BSZ *count*[,*value*]
: Reserve Zeroed Bytes. Inserts a sequence of *count* bytes of zero, or *value* if specified. The two-argument form is effectively the same as **FILL** with its arguments swapped.

  ZMB and **BSZ** are alternate forms recognised for compatibility with other assemblers.

Code placement & addressing:

ORG *address*
: Sets the Program Counter—the base address assumed for the next assembled instruction. Unless followed by a **PUT** pseudo-op, this will also be the instruction's actual address in memory. A label on the same line will define a symbol with a value of the specified address.

PUT *address*
: Modify the put address—the Program Counter is unaffected, so the assumed address for subsequent instructions remains the same, but the actual data will be located elsewhere. Useful for assembling code that is going to be copied into place before executing.

RMB *count*
: Reserve Memory Bytes. The Program Counter is advanced *count* bytes. In some output formats this region may be padded with zeroes, in others a new loadable section may be created.

SECTION *name*
CODE
DATA
BSS
RAM
AUTO Switch to the named section. The Program Counter will continue from the last value it had while assembling this section, or follow the previous section if had not previously been seen.

  Each of **CODE**, **DATA**, **BSS**, **RAM**, and **AUTO** switches to a section named after the pseudo-op. They are recognised for compatibility with other assemblers.

SETDP *page*
: Set the assumed value of the Direct Page (**DP**) register to *page* for subsequent instructions. Any non-negative *page* is truncated to 8 bits, or specify a negative number to disable automatic direct addressing.

  See the section on Direct Page addressing for more information.

Symbols:

EQU *value*
: Short for “equate”, this must be used with a label and defines a symbol with the specified *value*. This may be any single valid argument (e.g. an expression or a string).

EXPORT *name*[,*name*]…
: Each *name*—either the name of a macro or a symbol—is flagged to be exported. Exported macros and symbols will be listed in the symbols output file, if specified.

SET *value*
: Similar to **EQU**, this must be used with a label and defines a symbol with the specified *value*. Unlike **EQU**, you can use **SET** multiple times to assign different values to the same symbol without

error.

Files:

END [*address*]
Signifies the end of input. All further lines are disregarded.

Optionally specifies an EXEC address to be included in the output, where supported by the output format. An EXEC address specified on the command line will override any value specified here.

INCLUDE *filename*
Includes the contents of another file at this point in assembly. The *filename* argument must be a string, i.e. delimited by quotes or **/** characters.

INCLUDEBIN *filename*
Includes the binary data from *filename* (which, as with **INCLUDE** must be a delimited string) directly.

Direct Page addressing
The 6809 extends the zero page concept from other processors by allowing fast accesses to whichever page is selected by the Direct Page register (**DP**). An assembler is not able to keep track of what the code has set this register to, but the information is useful when deciding which addressing mode to use for an instruction. The **SETDP** pseudo-op, or **--setdp** option, informs the assembler that the supplied value is to be assumed for **DP**. Set this to a negative number to undefine it and disable automatic use of direct addressing (this is the default).

LICENCE