The as12 assembler

Modified: 8 July 1996 by Karl Lunt

Last Modified: 7 April 1999 by Tom Almy

Introduction

The as12 assembler is a two-pass cross-assembler for the Motorola 68hc12 microcontroller (MCU). as12 was written in C and derived from the source code for the original asm11 68hc11 assembler. So far, as12 has been ported to Sun Sparcstations, HPs, IBM PC-compatibles, and Apple Macintoshes.

as12 was designed to run from a command-line interpreter (CLI), so it works well with DOS batch files and Unix shell scripts. The Mac version of as12 has been integrated with the Macintosh Programmer's Workbench (MPW).

Unless otherwise noted, this document describes the PC version of as12.

Additional changes by Tom Almy corrects documentation, adds a unary complement operator, and provides a Windows console mode compiler which handles long file names.

---

as12 constructs

Invoking the as12 assembler

To start the as12 assembler, enter the following command at the prompt:

```
as12 file.ext
```

where file.ext is the path, name, and extension of the file you want to assemble. The as12 assembler will assemble the file, sending the listing output to the console and writing the S19 object output to a file named m.out. To save the listing output in a text file for later review, use the -L option. For example:

```
as12 foo.asm -Lfoo.lst
```

will assemble the file foo.asm and write the output listing to a file named foo.lst.

Entering **as12** with no arguments will display a short help file describing the available command-line options.

Command-line Options

Command-line options allow you to specify input file names, define global symbols, and declare paths for library files. For example:

```
as12 -dTIMERS -l\mylib\hc12 foo -L
```

will assemble the file foo.asm, using the library files found in the directory \mylib\hc12. Additionally, the label TIMERS will be defined with a value of 1 and the listing output will be written to file foo.lst.

When specifying the assembler source file, the extension ".asm" is assumed if no extension is explicitly given. If more than one source file is specified, then they will be read in the order listed.

The full set of command-line options includes:

- -d Define a label
- -l Define a library path
- -p Define a target processor type
- -D Turn on debugging output
- -L Specify a list file

-d Define a label

The -d option allows you to define a label on the command line. Labels defined in this manner are treated by the as12 assembler as if they had been defined within your source file and assigned a value of 1. Your source file can then refer to these labels in #ifdef tests. For example:

as12 -dMY_ALGORITHM myprog.asm

causes the as12 assembler to behave as if your source file began with the line:

#define MY_ALGORITHM

This ability to define labels from the command line adds great power to the as12 assembler. You can use this feature to selectively assemble blocks of source code based on arguments you specify in the command line, without first having to edit the source code before each assembly.

-l Define a library path

Normally, as12 first checks in the current directory for needed source and include files. If as12 cannot find a needed file in the current directory, it then checks the path for a previous source file and searches that directory. If that search also fails, as12 will use the library path specified with the -l option on the command line, if any. For example:

```
as12  -lc:\mypath  myfile.asm
```

---

define part string

as12 allows you to pass in a special string which will identify the part you are compiling the program for. When combined with the #ifpart conditional assembly directive, can give users a powerful way to compile source code which may depend upon what part is being targete d.

An illustration of it usage...

```
as12 -pb32
```

---

debug

Turns on the internal debug features of as12. Mostly for the developer of as12, but may help you if you are having a problem.

```
as12 -d source_file
```

---

listing

Specifies a listing file. If no filename extension is given, ".lst" is assumed. If no file name is given, then the file name will be that of the first source file, with an extension ".lst".

---

help

This help is designed as a reminder to the other command line options.

Example...

```
as12 -h
```

Pound Sign (#) Operators

- #include
- #define
- #ifeq
- #ifneq
- #ifdef
- #ifpart
- #else
- #endif
- Typical Conditional Assembly Examples

#include

The include directive allows other assembly source files to be inserted in the code immediately after the include statement, *as if* the c ontents of the included file were actually in the file that contained the include statement. Stated differently, the include statement works as you m ight expect. The syntax of the include statement is shown below...

```
#include /my_dir/myfile.s
```

It is important to note that the filename expansion will only be as good as the filename expansion as the shell that you are operating in. For ex ample, if you are running shell (/bin/sh) then the tilde username (~user) lookup may not work correctly. It is best to put in relative or absolute fi lepath specifications that are not shell dependent.

include statements may be used within #ifdef statements.

#define

The define statement allows labels to be defined. This statement is simply an alternate form for an equ assembler directive. T he alternate form is provided so that users will be alerted to the opportunities to write more sophisticated code that the #ifeq, and related statements allow. The proper use of the define statement is...

```
#define MY_LABEL expression
```

The define statement is as if the user had typed the following...

```
MY_LABEL  EQU  expression
```

Both forms are equally valid, and both forms are implemented internally thee same way. The EQU is probably more portable of the two constructs.

#ifeq

The ifeq command allows for the user to conditionally compile different sections of assembly language based on whether or not a label is equal to a value. Example...

```
#ifeq MY_SYMBOL  expression_to_compare_to
        ... (this code will be executed if MY_SYMBOL has the same value as
the expression_to_compare_to)
        ...
#endif
```

I show the #endif statement because for every form of *if* there needs to be a marker so that as12 knows what code is to be conditionally compiled. Restated, for every if there needs to be an endif.

If the expression resolves to the same value as the label in an #ifeq directive, then every line between the #ifeq and the #endif is executed. If the expression resolves to a different value than the label, all of the lines between the #ifeq and the #endif are ignored.

#ifneq

The ifneq command allows for the user to conditionally compile different sections of assembly language based on whether or not a label is not equ al to a value. Example...

```
#ifneq MY_LABEL  expression_to_compare_to
        ... (this code will be executed if MY_LABEL has a different value
as the expression_to_compare_to)
        ...
#endif
```

I show the #endif statement because for every form of *if* there needs to be a marker so that as12 knows what code is to be conditionally compiled. Restated, for every if there needs to be an endif.

If the expression resolves to the same value as the label in an #ifeq directive, then every line between the #ifeq and the #endif is ignored. If the expression resolves to a different value than the label, all of the lines between the #ifeq and the #endif are executed.

#ifdef

The ifdef command allows for the user to conditionally compile different sections of assembly language based on whether or not a label is defined (via a #define or an EQU. Example...

```
#ifdef MY_LABEL
        ... (this code will be executed if MY_LABEL has been defined)
        ...
#endif
```

I show the #endif statement because for every form of *if* there needs to be a marker so that as12 knows what code is to be conditionally compiled. Restated, for every if there needs to be an endif.

If the label in an #ifdef directive is defined, then every line between the #ifeq and the #endif is executed. If the label is not defined, all o f the lines between the #ifdef and the #endif are ignored.

#ifpart

This is the only directive that allows for a string comparison. A special internal variable is the only variable which is a string variable. The only way to set that variable is with the -p command line option. The sole purpose of this directive is to allow for conditional as sembly based upon the value of the string. This seemed natural for handling the different part types. Example...

```
#ifpart b32
  ... (assembly code) (will be executed if the string <b32> is same as
string in -p option
  ...
#endif
```

I show the #endif statement because for every form of *if* there needs to be a marker so that as12 knows what code is to be conditionally compiled. Restated, for every if there needs to be an endif.

If the string that follows the #ifpart directive matches the string that was passed in via the -p option, then the lines betwee n the #ifpart and the #endif will be executed. If the strings do not match, the lines between the #ifpart and the #endif will be ignored.

#else

This directive must be coupled with any of the if directives. This allows either or compilation and performs just like you expect an else to perf orm. Example...

```
#ifdef MY_LABEL
        ... (assembly code) (will be executed if MY_LABEL is defined)
#else
        ... (assembly code) (will be executed if MY_LABEL is NOT defined)
#endif
```

I show the #endif statement because for every form of *if* there needs to be a marker so that as12 knows what code is to be conditionally compiled. Restated, for every if there needs to be an endif.

If the if statement that goes with the else statement is true, the statements between the if and the else will be assembled, and the statements b etween the else and the endif will be ignored. If the if statement is false, the statements between the if and the else will be ignored and the state ments between the else and the endif will be executed.

There can only be one else for each if statement.

#endif

The endif statement tells the assembler when the conditional assembly section of the code is finished. Otherwise the assembler would have no way of knowing when to quit.

For every if statement there needs to be one endif. If there is an if and an else, then there should be one end statement also.

Examples...

```
#ifpart part_name
#else
#endif

#ifndef MY_LABEL
#endif

#ifndef MY_LABEL
#else
#endif
```

Typical Conditional Assembly Examples

- Use to handle parts starting in different modes. You can automate this and keep from modifying your source code my defining the label by invokin e the assembler using the -d command line option.

```
#ifdef EXPANDED_MODE
        org START_OF_EXTERNAL_RAM_TESTS
#else
        org     START_OF_FLASH_RAM
#endif
```

-

Use to handle configuring software so that your code will operate regardless of what part might be used. You can keep from changing your source code by passing in the parttype using the -p command line option.

```
#ifpart b32
RAM_START       EQU     $800
FEE_START       EQU     $8000
REG_START       EQU     $0000
PWM_START       EQU     $c7
#endif

#ifpart a4
RAM_START       EQU     $600
REG_START       EQU     $0100
#endif
```

Notice how easy you could build a library of different parts and make your source code compile accordingly.

Files

- as12.exe
- ?.s19
- the source file(s)
- the listing

---

as12.exe

This executable file was created with Borland C++ version 5.0, targeting the Win32 Console.

?.s19

A file with the same name as the first source file but with the extension ".s19" is always produced by the assembler. It cannot be suppressed. It is the s-records that are created by assembling the source file that is given to as12 on the as12 command line.

99 times out of 100 times, this is the file of interest when using the assembler.

For information regarding s-records (like a spec, but not quite) go my little s-record description.

---

the listing

The listing file is useful for debugging. Simply add the command line option "-L" to create the listing file.

---

the source file(s)

Standard ASCII source files. These should be created with the extension ".asm" since that is the default used by the assembler.

Features

AS12 Directives (or pseudo-opcodes)

- bsz
- db
- dc.b
- dc.w
- ds
- ds.b
- ds.w
- dw
- end
- equ
- fcb
- fcc
- fdb
- fill

- loc
- nam
- name
- opt
- org
- pag
- page
- redef
- rmb
- rmw
- spc
- ttl
- zmb

Expressions

Expressions may consist of symbols, constants or the character '*' (denoting the current value of the program counter) joined together by one of the operators: +-*/%&|^. You may nest expressions using parentheses up to 5 levels deep. The operators are the same as in C:

```
+       add
-       subtract
*       multiply
/       divide
%       remainder after division
&       bitwise and
|       bitwise or
^       bitwise exclusive-or
```

In addition, the unary minus (-) and complement (~) operators are allowed when preceding a symbol, constant, or character '*' only.

```
Examples of valid expressions...

(5*8)
(my_val-10+20*(16-label)/10)
10
$10
*
%10010
my_value
~$20
```

Note: When the asterisk (*) is used in a context where the as12 is expecting a label, the asterisk (*) represents the value of the current program counter.

Symbols

Symbols consist of one or more characters where the first character is alphabetic and any remaining characters are alphanumeric. Symbol are case sensitive.

Constants

Constants are constructed with the same syntax as the Motorola MDOS assembler (oh, now thats a real useful piece of information - hey I just copied this anyway):

```
'       followed by ASCII character
$       followed by hexadecimal constant
@       followed by octal constant
%       followed by binary constant
digit   decimal constant
```

Labels

A symbol starting in the first column is a label and may optionally be ended with a ':'. A label may appear on a line by itself and is then interpreted as:

```
        Label   EQU     *
```

Note that labels are case sensitive. "Label" and "label" are different labels.

Comments

Here are some notes about comments...

- Any line beginning with an * is a comment
- Any line beginning with a ; is a comment
- *You must* have a ; (semi-colon) prefixing any comment on a line with mnemonics

---

The bsz PSEUDO OPCODE

Name: Block Set Zeros

Description:

Zeros memory. Please use zmb instead.

Actually, don't use these at all.

The db PSEUDO OPCODE

Name: Define Byte

Syntax and examples (maybe):

```
        db      Byte_Definition[,Byte_Definition]

        db      $55,$66,%11000011
        db      10

half    db      0.5*100
```

Description:

Defines the value of a byte or bytes that will be placed at a given address.

The db directive assigns the value of the expression to the current program counter. Then the program counter is incremented.

Multiple bytes can be defined at a time by comma separating the arguments. Each comma separated argument can be a separate expression that the as 12 will evaluate.

Notes:

- This is probably a more universally accepted pseudo-op than the fcb. However, the selection of a pseudo op does have implications on portability. I provide as many as I can to enhance OUR ability to read other peoples code.
- This should be used for memory that is not considered volatile (ROM/EE/FLASH) or memory that will be boot-loaded or similar. For defining RAM memory for variables and scratchpad memory the ds directive is more appropriate.

Related To:

- fcb
- fdb
- dw
- ds

Useful With:

- Defining Data Tables/Structures
- Defining ASCII phrases (strings)
- Defining Constants

Things to look out for:

- Be careful not to define values that are *larger* than 8 bits. as12 truncates the left most bits to make the byte fit into a byte.
- A label is usually used so there is a reference to this memory. In the last example in the Syntax section, it can be seen that the label *half* will refer to the byte with a decimal value of 50. (Not really fixed point math but I'm only demonstrating the use of a label)

The dc.b PSEUDO OPCODE

Name: Define Constant Byte

Description:

Identical to db. My preference is to use the db and not this one. This is only to help read other peoples software that may get sent to us.

The dc.w PSEUDO OPCODE

Name: Define Constant Word

Description:

Identical to dw.

The ds PSEUDO OPCODE

Name: Define Storage

Syntax and examples (maybe):

```
ds      Number_of_Bytes_To_Advance_Program_Counter
```

Description:

The ds increments the program counter by the value indicated in the Number of Bytes argument.

Notes:

- This is the preferred method of defining a memory location whose value...
  - is changing
  - is generally not known
- In other words, this is optimal for defining RAM or REGISTER spaces. The reason for this is the ease in which a ds based region can be relocated .

Related To:

- rmb

Useful With:

- RAM definitions
- REGISTER definitions

Things to look out for:

- Inappropriate for non-volatile memory definitions

The ds.b PSEUDO OPCODE

Name: Define Storage Bytes

Description:

Identical to ds. I don't care what form that you use, but I imagine that the ds is better than the ds.b. I hope you don't mind *clicking* one more time to get to the right spot in the manual.

The ds.w PSEUDO OPCODE

Name: Define Storage Word

Syntax and examples (maybe):

```
ds.w    Number_of_Words_To_Advance_Program_Counter
```

Description:

The ds.w increments the program counter by the value indicated in the Number of Words argument multiplied by two. In other words, if the ds.w expression evaluates to 4 then the program counter is advanced by 8.

Notes:

- Good for defining RAM and REGISTERS

Related To:

- ds

Useful With:

- labels

Things to look out for:

- Inappropriate for non-volatile memory.

The dw PSEUDO OPCODE

Name: Define Word

Syntax and examples (maybe):

```
        dw      Word_Definition[,Word_Definition]

        dw      $55aa,$66,%11000011
        dw      10

half    dw      0.5*65536
```

Description:

Defines the value of a word or words that will be placed at a given address.

The dw directive assigns the value of the expression to the current program counter. Then the program counter is incremented by 2.

Multiple words can be defined at a time by comma separating the arguments. Each comma separated argument can be a separate expression that the as 12 will evaluate.

Notes:

- This is probably a more universally accepted pseudo-op than the fdb. However, the selection of a pseudo op does have implications on portability. I provide as many as I can to enhance OUR ability to read other peoples code.
- This should be used for memory that is not considered volatile (ROM/EE/FLASH) or memory that will be boot-loaded or similar. For defining RAM memory for variables and scratchpad memory the ds directive is more appropriate.
- Words are right justified and left filled with zero's.

Related To:

- fdb
- dc.w

Useful With:

- Defining Data Tables/Structures
- Defining Constants

Things to look out for:

- Be careful not to define values that are *larger* than 16 bits. as12 truncates the left most bits to make the word fit into a word.

The end PSEUDO OPCODE

Name: End

Description:

Identical to ttl.

The equ PSEUDO OPCODE

Name: Equate

Syntax and examples (maybe):

```
Label   EQU     Value_To_Assign_To_The_Label
```

Description:

Directly assigns a value to a label.

Notes:

- Very good for constants
- Most common across different assemblers (most likely to port easily)

Related To:

- #define
- -d command line option

Useful With:

- #ifeq and related options

Things to look out for:

- Be careful of how many bits your label can take. The as12 internally uses anywhere from 32 bits for the label value with the Win32 version. It is very easy to get bigger than 8 or 16 bits.
- Inappropriate for defining memory locations. I would recommend only using for defining constants. Otherwise relocation can be made very difficult.

The fcb PSEUDO OPCODE

Name: Form Constant Byte

Description:

Identical to db. The db is the preferred command. This flies in the face of Motorola history, but I believe that *externally* our files may be more compatible (Although I can't prove it so do what you want).

The fcc PSEUDO OPCODE

Name: Form Constant Characters

Syntax and examples (maybe):

```
fcc     delim_character<i>string_to_encode</i>delim_character
fcc     /my_string/
fcc     *// string with slashes //*
fcc     'best to use single quotes'
```

Description:

FCC allow the encoding of a string.

The first character is the delimiter. By allowing the flexibility of selecting delimiters, you can easily make strings which have slashes and tick marks in them. The *only* catch is that if you choose a delimiter, it

- must also be used to mark the end of the string
- it cannot appear in the string as a character.

In the second example, my_string will be encoded as an ASCII string. The /'s simply mark the ending and beginning of the string. This also lets you put spaces in the string.

In the third example, the * (asterisk) is the delimiter and the slashes will be encoded with their ASCII values into the ASCII string.

I like single quotes the best as a delimiter. You could argue that double quotes are even better because it follows 'C' convention.

Notes:

- You cannot have the space as a delimiter.
- I believe that you can have strings in the FCB except that you have to encode them one at a time and comma delimit them. Yuk.

Related To:

- fcb

Useful With:

- Defining strings for displays and such.

The fdb PSEUDO OPCODE

Name: Form Double Byte

Description:

Identical to dw. I prefer the dw usage.

The fill PSEUDO OPCODE

Name: Fill Memory

Syntax and examples (maybe):

```
        fill    byte_to_fill_memory_with,num_of_bytes_to_fill
```

Description:

FILL allows a user to fill memory with a byte. See my comments in zmb about the value of these pseudo opcodes.

Notes:

- Nice for initializing memory.

Related To:

- zmb

Useful With:

- Debugging
- Filling unused non-volatile memory with a *safe* opcode when the processor gets lost.

Things to look out for:

- Since RAM memory, by definition, cannot be initialized, this command has little use. This is because you must DOWNLOAD the s-records to make the clearing take place. Only in systems which have some sort of bootstrapping (where s-records are downloaded) would this be very useful. If you are clearing memory, you should probably count on routines to do it for you.

The loc PSEUDO OPCODE

Another Pseudo OP called LOC basically increments and produces an internal counter used in conjunctions with the backwards tick mark (`). By using LOC's and the ` mark you can do code like the following without worrying about thinking up new labels.

```
        LOC
        ldaa    #1
loop`
        deca
        bra     loop`
        LOC
loop`
        brset   0,x $55 loop`
```

This code will work perfectly fine because the second loops label is really loop002 and the first ones is loop001. The assembler really sees this...

```
        LOC
        ldaa    #1
loop001
        deca
        bra     loop001
        LOC
loop002
        brset   0,x $55 loop002
```

You may also seed the LOC with a valid expression or number by putting that expression or number in the operand field. This gives you the ability to over ride the automatic numbering. This is also sometimes handy if you need to keep track of what your local variable is. (you lose track in the source if you aren't careful, because the tick ' mark is the only thing you see).

The name PSEUDO OPCODE

Name: *N*ot *A*ny *M*nemonic

Description:

Ignored

The opt PSEUDO OPCODE

Name: Assembler Options

There are five permissible operands for this instruction:

- l - enable listing after *opt nol*
- nol - disable listing until *opt l* or end of source code
- c - calculate execution time (clock cycles)
- noc - stop calculating execution time
- contc - continue calculating execution time

The org PSEUDO OPCODE

Name: Origin

Syntax and examples (maybe):

```
org     value_to_set_program_counter_to

org     $800    ;not my preferred form
org     MY_PROGRAM_START        ;better form
org     LAST_MEMORY_LOCATION-(LAST_PROGRAM_BYTE-FIRST_PROGRAM_BYTE);best but complex and has reference problems
```

Description:

The org pseudo opcode allows the assembler's program counter to be set to a value. This is useful for locating your software and its elements (tables, ram, constants, etc) in useful (intelligent) locations within the memory space of the microcontroller.

In better multi-pass assemblers (not as12), the org statement is rarely used because the code is located at the link, and not during compilation. Since as12 is a *simple* two-pass assembler, orgs must be used so that the code is compiled where it is supposed to.

Notes:

- When starting a new region of code, you can examine the s-record file and see how org affects the construction of that file.
- It is better to use the form *org label* than *org constant* because the more constants that are buried within your code, the more difficult it is to reuse.
- The less orgs you use, the more reusable your code is.

Related To:

- program counter because this sets its value
- rmb and its cousins because they change the program counter

Things to look out for:

- Always find out where the orgs are in a program. This is the first key to understanding the program.

The pag PSEUDO OPCODE

Name: Poor Aging Gophers

Ignored

The redef PSEUDO OPCODE

Name: *Redef*ine

Used to redefine first operand (which must be a label) to value of second operand (an expression)

Example:

```
foo     equ     10
        ldaa    #foo        ; Accumulator A gets value 10
        redef   foo 12
        ldab    #foo      ; Accumulator B gets value 12
```

The rmb PSEUDO OPCODE

Name: *R*eserve *M*emory *B*ytes

Equivalent to *ds.b* or *ds,* which is preferred.

The rmw PSEUDO OPCODE

Name: *R*eserve *M*emory *W*ords

Equivalent to *ds.w.*

The spc PSEUDO OPCODE

Name: *Spac*e

Ignored

The ttl PSEUDO OPCODE

Name: *Titl*e

Ignored

The zmb PSEUDO OPCODE

Name: *Z*ero *M*emory *B*ytes

Operand specifies number of bytes to allocate and fill with zero. Use is not recommended.