A Short Guide to FFASM
flashforth assembler

ffasm is a slightly extended and enhanced version of Mikael Nordman's asm2.txt for AVR Atmega devices. See: https://github.com/oh2aun/flashforth/tree/master/avr/forth
It was written for use with Mikael Nordman's **flashforth** and been tested with other Forth implementations.

ffasm allows assembler code to be used within Forth words using a syntax almost identical to normal assembler rather then the reverse Polish notation of many other Forth assemblers. It is also compact requiring less than 2K bytes of flash.

Not all AVR instructions are currently implemented, but it is relatively trivial to add additional instructions provided they fit within the already defined rules. IMHO, compactness trumps completeness for small memory systems.

WARNING: There is no error checking regarding whether a number or register is outside the permitted range allowed for a particular instruction. **ffasm** masks parameters to the number of bits required for the specified instruction, so incorrect parameters will still compile. For example: The **ldi** instruction only works on registers R16-R31. The instruction **as: ldi r0 #9** will compile without error but the resulting code will be for: **ldi r16 #9**

Some trivial examples should illustrate how to use **ffasm** to create assembler words**:**

```
: ex1
  as: ldi r16 $ff       \ Load r16 with $ff
  as: begin             \ Start loop
  as:   dec r16         \ Decrement r16
  as: until eq          \ Leave loop when r16 equals zero [brne begin:]
;                       \ Return
```

All assembler instructions are prefixed by **as:**

The trailing semicolon results in a **ret** instruction.

The mnemonics for each instruction are the same as in the Atmel AVR Instruction Guide but always in lower case. Parameters are separated by white space. Where an instruction needs a register as a parameter they are referred to by **r0** to **r31.**

Examples: **add r1 r7 cpc r3 r9 cpse r24 r7**

Instructions which access configuration registers, set/clear/test bits, address memory or take immediate values can take literals, or consume a value from the stack.

Examples: **subi r24 1 adiw r30 #10 sbi $25 3 lds r24 $300**

To take a value from the stack use the ^ character.

Examples: **ldi r16 ^ sbiw r24 ^**

For instructions where two parameters are required and they are both going to be on the stack then the usual reverse Polish order will apply.

Example:
```
$25 constant PORTB      \ Define register as constant
3   constant LED        \ Define pin number as constant
: led_on                \ Word to turn on pin 3 of PORTB
  [ LED PORTB ]         \ Put the parameters on the stack
  as: sbi ^ ^           \ Set PORTB pin 3 to high.
;
```

It is essential to place the values on the stack within **[** and **]** immediately before the

instruction. This not only aids readability, it also ensures they do not interfere with the stack values left by flow control structures (see later). If a word contains no flow control structures it is possible to place them on the stack before starting the colon definition.

NOTE: A serial application, such as **forthtalk** that substitutes register names with literals during the upload makes for more readable and compact code. The same example would simply be:

```
: led_on
  as: sbi PORTB LED
;
```

However, it can still be useful to use the stack when, for instance, you need to calculate a register mask as in this code snippet for use with **forthtalk**:

```
[ SPE MSTR SPR0 or or ]
as: ldi r17 ^    \ Set SPE MSTR and SPR0 bits in r17
as: out SPCR r17 \ Load them into the SPI Control Register
```

It is also, of course, possible to calculate a value as in this example of a 50uS delay taking into account processor speed which is available in **flashforth** in the constant **Fcy**.

```
: 50uS
  [ Fcy 1000 /          \ Switch to interpreter to calculate cycles per 1uS
  50 *                  \ Calculate number of cycles to burn in 50uS
  3 / ]                 \ Calculate delay loops reqd then switch back to compile
  as: ldi r16 ^         \ Load r16 with number of cycles from stack
  as: begin             \ Each loop iteration takes 3 processor cycles
  as:   dec r16         \ Decrement r16 – 1 cycle
  as: until eq          \ brne instruction – 2 cycles
;
```

I/O instructions take the memory mapped value for registers. These are automatically adjusted to direct I/O addresses. Specifically **cbi, sbi, sbic** and **sbis** require a register reference in the range **$20 - $3f** and **in** and **out** require a register reference in the range **$20 - $5f**.

Examples:

```
sbi $25 3 (Set pin 3 of PORTB to high)
sbic $23 0 (Skip next instruction if pin 0 of PORTB is zero)
out $24 r24 (Set DDRB register with value in r24)
in r24 $23 (get the current value of all PORTB pins into r24)
```

Indirect addressing registers are referred to by: **x y z**
Indirect with post-increment is indicated by: **x+ y+ z+** and pre-decrement by: **-x -y -z**
Examples: **st -y r24 ld r16 x ld r24 x+ st -z r20**
Load or store indirect with displacement instructions such as: **ldd r24 y+q** or **std z+q r24** are not supported.

None of the standard **br*xx*** branch instructions are implemented. Branches are replaced with the flow control structures:

ffasm	**assembler**
if xx … then	**brxx then** … then: …
if xx … else … then	**brxx else** … **rjmp then** else: … then:
begin … until xx	begin: … **brxx begin** …
begin … while xx … repeat	begin: … **brxx exit** … **rjmp begin** exit:
begin … repeat	begin: … **rjmp begin**

which in combination calculate the appropriate **brxx** branches. The flow control words leave values on the stack which are used to calculate the branches and jumps.

xx can be any of the usual AVR branch mnemonics:

cc Carry Cleared	**lt** Less Than (Signed)
cs Carry Set	**mi** Branch if Minus
eq Equal	**ne** Branch if Not Equal
ge Greater or Equal (Signed)	**pl** Branch if Plus
hc Half Carry Flag is Cleared	**sh** Same or Higher (Unsigned)
hs Half Carry Flag is Set	**tc** T Flag is Cleared
id Global Interrupt is Disabled	**ts** T Flag is Set
ie Global Interrupt is Enabled	**vc** Overflow Cleared
lo Lower (Unsigned)	**vs** Overflow Set

If **begin** is paired with **repeat** but you must provide an 'exit' unless deliberately creating an infinite loop, e.g a **ret** within **if** and **then** as below:

```
: ex2
  as: ldi r16 $4        \ Load r16 with $4
  as: begin             \ Start loop
  as:   dec r16         \ Decrement r16
  as:   if ne           \ If not equal goto then [brne then:]
  as:     clr r25       \ Clear r25
  as:     ret           \ Return – same as the Forth word 'exit'
  as:   then
  as:   lsl r24         \ Logical shift left r24
  as: repeat                \ Loop back to begin [rjmp begin:]
;
```

Flow control can be nested but you should note that **br*xx*** branches can only reach +63 or -64 words (~=instructions) from the current location. Keep your definitions relatively short!

In some circumstances it may be possible to use one of the **sbrc**, **sbrs**, **sbic** or **sbis** instructions to skip over the **repeat** [**rjmp**] instruction:

```
: ex3
  as: ldi r16 $ef       \ Load r16 with $ef = %11101111
  as: begin             \ Start loop
  as:   lsr r16         \ Logical shift right r16
  as:   sbrc r16 0      \ Skip next instruction if Bit0 in r16 is cleared
  as: repeat            \ Loop back to begin [rjmp begin:]
;
```

flashforth has quite a few words which are always inlined, i.e. their code is inserted directly into a newly defined word rather than being accessed via a call to a subroutine.

These are: **rp@ >< cell+ cells char+ chars invert 1+ 1- 2+ 2- 2* 2/ p+ @p p2+ ei di dup drop rdrop >body idle busy**

These words are listed on the flashforth page: http://www.flashforth.com/atmega.html

Inlined words can therefore be intermixed with assembler with no overhead.

```
: ex5 ( n – n )
  dup                   \ Duplicate the top item on the stack
  di                    \ Disable interrupts
  as: ldi r16 #16       \ Load r16 with decimal 16
  as:  begin            \ Start loop
  as:    lsr r25        \ Shift ToSH right through C
  as:    ror r24        \ Rotate through C ToSL
```

```
  as:    if cs
  as:      sbi 5 3      \ Set Port B pin 3 high if C set
  as:    then
  as:    if cc
  as:      cbi 5 3      \ Set Port B pin 3 low if C clear
  as:    then
  as:    dec r16        \ Decrement r16
  as: until eq          \ Leave loop when r16 equals zero [brne begin:]
  ei                    \ Enable interrupts
  drop                  \ Delete ToS
;
```

Some other **flashforth** words can be inlined by prefixing them with the Forth word **inline** namely: **ticks 1 over swap + - and or xor mset mclr lshift rshift sp@ sp! !p p++ flash eeprom ram cell false true state ticks >pr d+ d2/ dinvert fl- fl+**

Of course any assembler words you define can also be marked to be automatically inlined by using the Forth word **inlined** after the semicolon.

```
: ex6 ( s8 – s16+1 )
  as: sbrc r24 7        \ Sign extend an 8 bit value to 16 bits
  as: ldi r25 $ff
  as: adiw r24 $01       \ And increment
; inlined
```