

# A Code Walk through Myforth-Arduino

---

Myforth for the Arduino is loosely based on myforth for the 8051. The 8051 version was an experiment in 8 bit Forth, which suited the 8051 nicely. The AVR instruction set doesn't lend itself to an efficient 8 bit implementation, in my opinion, so it has 16 bit stacks. It still has other non-standard features. For example conditionals don't pop the stack and ; can change the previous call into a jump.

We'll start the walk-through at the top, with the bash script named "c".

```
gforth ./job.fs -e "bye "
```

The first tool needed then is Linux and bash. I have used this compiler with Babun in Windows, though I haven't yet tried downloading there. Typing "./c" at the command line in bash invokes gforth, the second tool we'll be needing for this project.

## job.fs (plus ansi.fs and vtags.fs)

Gforth interprets the bash command line. The "-e" causes the string "bye" to be executed after the file "./job.fs" has been included. By convention then, "job.fs" is the name of our main load file.

Each of the files begins with a copyright notice. I chose the lesser GPL in the hope that commercial users would have the option of protecting their source code if they wanted to. I'm not sure that I understand correctly how this works, and I'd be glad to have someone inform me if I'm wrong.

```
24 only forth also definitions
25 : nowarn warnings off ; : warn warnings on ; : not 0= ;
26 nowarn
27
28 \ Colors are used by the decompiler/disassembler
29 include ansi.fs \ Part of Gforth.
30 warn
31 variable colors
32 : in-color true colors ! ;
33 in-color
34 : b/w false colors ! ;
35 : color ( n - ) create , does> colors @ if @ >fg attr! exit then drop
36 ;
37 red color >red
38 black color >black
39 blue color >blue
40 green color >green
41 cyan color >cyan
42 yellow color >yellow
43
44 \ For navigating the source code in the VIM editor
45 include ../vtags.fs use-tags
```

Note the syntax coloring. It comes from my custom syntax file for vim, the third tool needed for this project. I made the coloring resemble colorForth as much as possible.

We start by setting the search order to forth and defining some mostly cosmetic things. We turn off warnings about redefinitions, load gforth's ansi terminal file for its colors, and compile some code that makes a vim tags file to help us navigate later. This has little to do with compiling for the Arduino so we won't go into it any further.

```
46 0 constant start \ Reset vector.
47 $68 constant rom-start \ Start of code, all interrupts reserved.
48 $8000 constant target-size
49
50 \ Initial stack pointers
51 $08ff constant r0
52 $06ff constant s0
53 nowarn \ warnings off
54 s0 1 + constant tib \ terminal input buffer
55 warn \ warnings on
```

Next we define some things about the chip we're using. "start" is the starting address of the program flash in this chip. Out of old habit we call it ROM, and define "rom-start" at hex 68, which is the end of the interrupt vector table. "\$8000" is the "target-size", 32K for the ATmega328. Each of the two stacks grows down in internal RAM. A terminal input buffer grows up from the top of the data stack. In the 8051 it was easier to have the two stacks go toward each other.

```
57 include ../ATmega328.fs \ Special Function Registers
```

## ATmega328.fs (Special Function Registers)

The file "ATmega328.fs" contains a list of the special function registers in the ATmega328, followed by definitions for "PORT", "DDR", and "PIN"; which help imitate the Arduino IDE later by identifying port pins by their Arduino number.

Here are just a few of the special function registers and PORT, DDR, and PIN. DDR means Data Direction Register.

```
55 $c6 constant UDR0 \ USART I/O Data Register
56 $c5 constant UBRR0H \ USART Baud Rate Register High
57 $c4 constant UBRR0L \ USART Baud Rate Register Low
58 $c2 constant UCSR0C
59 $c1 constant UCSR0B
60 $c0 constant UCSR0A
61
62 : PORT ( n - bit port) dup 8 < if PORTD exit then -8 + PORTB ;
63 : DDR ( n - bit port) dup 8 < if DDRD exit then -8 + DDRB ;
64 : PIN ( n - bit port) dup 8 < if PIND exit then -8 + PINB ;
```

The AVR has 32 8 bit registers, three pairs of which work together as 16 bit index registers. We name them X, Y, and Z which is the AVR standard naming. The top of the data stack is cached in the register pair named T, and N is a temporary cache for the second stack item on the data stack.

```
59 30 constant Z 31 constant Z' \ used as loop counter
60 28 constant Y 29 constant Y' \ address register
61 26 constant X 27 constant X' \ pointer to rest of stack
62 24 constant T 25 constant T' \ top of stack
63 22 constant N 23 constant N' \ next on stack (temporary)
```

The next four lines load the target compiler, disassembler, assembler, and finally the Forth primitives for the AVR.

```
65 include ../compiler.fs
66 include ../disAVR.fs
67 include ../asmAVR.fs
68 include ../miscAVR.fs
```

Let's start with the target compiler.

## compiler.fs (Target Compiler)

```
24 only forth also definitions
25 vocabulary targ
26
27 nowarn
28 : target only forth also targ also definitions ; immediate
29 : ] postpone target ; immediate
30 : host only targ also forth also definitions ; immediate
31 : [ postpone host ; immediate
32 host
33 warn
```

We're going to want to have new definitions for a number of gforth words and we don't want to lose the originals, so we make a vocabulary called "targ". The words "target" and "host" change the search order and have aliases called [ and ] for convenience.

```
35 : :m postpone target : ;
36 : m; postpone ; postpone host ; immediate
```

Soon both ":" and ";" will be redefined, and these "m" versions allow us to keep the old behaviors as well.

```
38 \ 0 constant start \ Reset vector.
39 \ 8192 constant target-size
40 create target-image target-size allot
41 target-image target-size $ff fill \ ROM erased.
42 : there ( a1 - a2) target-image + ( start + ) ;
43 : c!-t ( c a - ) there c! ;
44 : c@-t ( a - c) there c@ ;
```

```

45 : !-t ( n a - ) there over 8 rshift over 1 + c! c! ;
46 : @-t ( a - n ) there count swap c@ 8 lshift + ;

```

Here we make space in the host for a target image. The size of the target image was determined earlier in job.fs. An erased flash is full of \$ff, so that's what we fill the image with. Target compiling boils down pretty much to storing instructions in the target image, which is done with these special ! and c! words. We also have words for fetching from the target image. The word "there" calculates a host address from a target address, and the -t on the end of these words means "there", as in "c@ from there". Note that !-t uses byte addresses even though the instruction set uses word addresses. Maybe that should be rethought.

```

48 variable tdp \ Rom pointer.
49 :m HERE ( - a ) tdp @ m;
50 :m ORG ( a - ) tdp ! m;
51 :m ALLOT ( n - ) tdp +! m;
52 :m , ( n - ) HERE !-t 2 ALLOT m;
53 : ,-t ( n - ) target , m;

```

As we compile code (and data) into flash (ROM) we keep track of the next available space with the Target Dictionary Pointer, "tdp". Now the target compiler has its own versions of "here", "allot", ",", and "-t", as well as "org" for setting the tdp.

```

55 variable trp \ Ram pointer.
56 : cpuHERE ( - a ) trp @ ;
57 : cpuORG ( a - ) trp ! ; 8 cpuORG
58 : cpuALLOT ( n - ) trp +! ;
59 : report cr ." HERE=" target HERE host u. cr ;

```

The AVR has separate addressing space for internal RAM, so there are separate locating words for that space, prefixed with "cpu". The "trp" RAM pointer is used to allocate variables in internal RAM.

```

61 \ ----- Optimization ----- /
62 variable 'edge
63 : hide target-size 1 - 'edge ! ; hide
64 : hint target here host 'edge ! ;
65 : edge 'edge @ ;

```

Finally, as in colorForth, a few instruction phrases can be optimized, and the words "hide" and "hint" help to control this by keeping track of or hiding the last compiled instruction.

```

67 \ ----- Labels ----- /
68 nowarn
69 variable labels 0 labels !
70 warn
71 : label ( - )
72   [ labels @ here labels ! , ] HERE host , BL word count here
73   over char+ allot place align ;
74 : show ( a - ) 2 cells + count type ;
75 : label? ( a - 0|a)
76   >r labels begin @ dup while dup cell+ @ r@ = if
77   r> drop exit then repeat r> drop ;
78 nowarn

```

```

79 : (words words ;
80 : .words labels begin @ dup while dup cell+ @ u. dup show 2
81   spaces repeat drop ;
82 : target-words
83   labels begin @ dup while dup show space repeat drop ;
84 warn

```

Rather than use a gforth vocabulary, a linear linked list of labels is created for the target words which can easily be displayed or searched. This is totally on the host, separate from the target resident headers.

```

86 \ ----- Headers on the target ----- /
87 variable thp
88 create target-heads target-size allot
89 create end-of-heads
90 : headsize end-of-heads thp @ - ;
91 target-heads target-size + 3 - thp !
92 0 value heads
93 nowarn
94 : header ( - )
95   thp @ >r labels @ cell+ dup cell+ dup c@ 3 + dup 1 and + negate thp +!
96   thp @ over c@ 1 + dup 1 and + move @ 2/ dup 8 rshift r@ 1 - c! r> 2 -
97   c! ;
98 warn
99 \ Tack headers onto end of code.
100 : headers ( - )
101   target-size target here host headsize + - 0<
102   abort" Target memory overflow"
103   thp @ target here host dup to heads there headsize move
104   headsize tdp +! ;

```

In order to have an on board interpreter (though not an on board compiler) we build a dictionary downward in program memory from the top. Later this will be appended to the end of the target dictionary. Myforth for the 8051 had short, four byte names to save memory, but for this version we have long names.

```

105 0 value save-fid
106 : save ( - )
107   0 to save-fid s" chip.bin" delete-file drop
108   s" chip.bin" r/w create-file abort" Error creating chip.bin"
109   to save-fid target-image target-size
110   save-fid write-file abort" Error writing chip.bin"
111   save-fid close-file abort" Error closing chip.bin" ;

```

Once the program is finished compiling we save its binary image as “chip.bin” to be downloaded later.

## disAVR.fs (disassembler)

Like myforth for the 8051, this one has a disassembler, and it was written mostly before but at least concurrently with the assembler. I used it to debug the assembler and now I use it to check my code for correctness and efficiency.

I don't want to show too much of the code because it's ugly and tedious, but it gets the job done. Let me describe it and show just a little bit instead.

```
224 nowarn
225 : show-name ( a) >red label? dup if 5 spaces dup show cr then
226   drop >black ;
227
228 : decode ( adr) cr 2*
229   begin dup show-name
230     dup >yellow 2/ .xxxx dup @-t dup .xxxx space >green .dis
231     key 13 - while 2 + repeat >black drop ;
232
233 : see also targ ' >body @ 2/ decode previous ;
234 warn
```

The word “see” looks up the address of a target word and passes this address to “decode”, which does the scanning and formatting. “decode” does a carriage return and “show-name” if there is a label at this address. Then the address and the 16 bit instruction at that address are displayed in yellow followed by the disassembly in green which we’ll look at in a bit. This repeats at each key press until a carriage return is pressed.

“decode” passes the current instruction to “.dis”, which looks like this:

```
203 : .dis ( instruction)
204   dup inst ! 12 rshift
205   dup $0 = if drop .x0 exit then
206   dup $1 = if drop .x1 exit then
207   dup $2 = if drop .x2 exit then
208   dup $3 = if drop .x3 exit then
209   dup $4 = if drop .x4 exit then
210   dup $5 = if drop .x5 exit then
211   dup $6 = if drop .x6 exit then
212   dup $7 = if drop .x7 exit then
213   dup $8 = if drop .x8 exit then
214   dup $9 = if drop .x9 exit then
215   dup $a = if drop .xa exit then
216   dup $b = if drop .xb exit then
217   dup $c = if drop .xc exit then
218   dup $d = if drop .xd exit then
219   dup $e = if drop .xe exit then
220   dup $f = if drop .xf exit then
221   drop ;
```

That current instruction is saved in a variable (inst) then shifted right by 12 places to isolate the top four bits, which is fed to a case statement to execute one of 16 little disassemblers. Each of these features some bit masking, shifting, and comparing to decide what to display. Here are some examples.

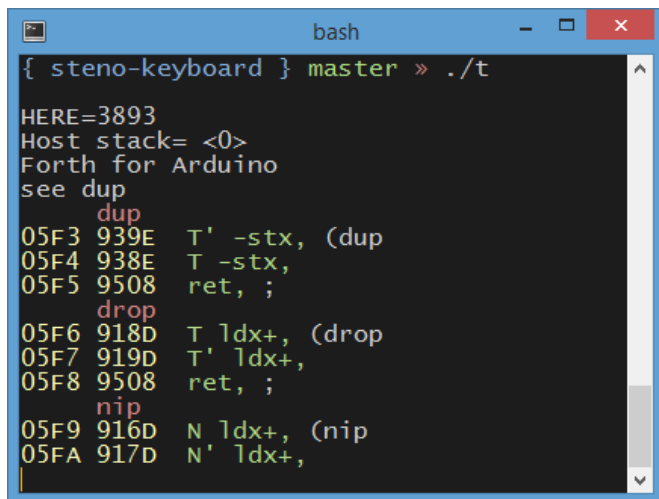
```
59 : .Rd inst @ 4 rshift $1f and .reg ;
60 : .RrRd inst @ dup $0f and swap $200 and 5 rshift or .reg .Rd ;
61 : .rel inst @ 3 rshift $3f and dup $20 and if
62   $3f invert or then dec. ;
63 : .long ( a1 - a2) 2 + dup @-t cr 11 spaces .xxxx ;
64 : .iw inst @ dup $0f and swap 2 rshift $30 and or dec.
```

```

65   inst @ 3 rshift $1e and 24 + .reg ;
66 : .bit inst @ 7 and dec. ;
67 : .bitIO .bit inst @ 3 rshift $1f and .io ;
68
69 : match ( opcode mask - flag) inst @ and = ;
70 : exact ( opcode - flag) >black inst @ = ;

```

The final product looks something like this:



```

bash
{ steno-keyboard } master » ./t
HERE=3893
Host stack= <0>
Forth for Arduino
see dup
  dup
05F3 939E T' -stx, (dup
05F4 938E T' -stx,
05F5 9508 ret, ;
  drop
05F6 918D T ldx+, (drop
05F7 919D T' ldx+,
05F8 9508 ret, ;
  nip
05F9 916D N ldx+, (nip
05FA 917D N' ldx+,

```

That's enough about that.

## asmAVR.fs (Just Enough of an Assembler)

Myforth for the 8051 had no assembler at all. It was an imitation of colorForth, which also has no assembler. Everything was done with hex machine codes. For the Arduino I wanted to be able to use assembly language easily so enough assembler was made to allow this.

First come the calls and returns.

```

24 :m interrupt ( adr) 2* 2 + target here host 2/ swap !-t m;
25 :m reti, $9518 ,-t m;
26
27 :m rel ( adr - n) here - 2/ 1 - m;
28
29 :m rcall, ( a) rel $0fff and $d000 or ,-t m;
30 :m lcall, ( a) $940e ,-t 2/ ,-t m;
31 :m call, ( a) hint dup here - abs $1fff < if
32   rcall, exit then lcall, ;

```

The interrupt word takes the word address of an interrupt vector and stuffs the current “here” after the jump instruction already there. The code that follows will be the interrupt routine. “reti,” compiles a return from interrupt. Note that by convention an assembler word ends with a comma. Note also that the address fed to “interrupt” is the word address of the interrupt vector “rel” is a helper that calculates a relative address; relative to here, the next available location program memory. Many of the jump instructions have both long and relative versions.

“rcall” changes the given absolute address to a relative address and compiles a call. A relative call or jump takes just one word of program memory. A long call or jump takes two words. AVR program memory is 16 bit word addressed. “lcall” assembles a call using the absolute or long address.

For convenience, the word “call” will assemble a relative call if it can, but will assemble a long call otherwise.

```
34 :m rjmp, ( a) rel $0fff and $c000 or ,-t m;
35 :m ljmp, ( a) $940c ,-t 2/ ,-t m;
36 :m jump, ( a) ljmp, m;
```

“rjmp,” “ljmp,” are similar but “jump,” is always a long jump and I don’t remember why. I’ll need to investigate that.

```
38 :m entry here dup 0 org ljmp, org m;
```

“entry” marks the current address as the entry point by storing the current address at the reset vector, address 0 in program memory.

```
40 :m -: host >in @ label >in ! create
41   target here host , hide postpone target
42   does> @ ( talks @ if talk exit then) target call, m;
43
44 :m : -: header m;
```

“-:” makes a new entry in the target compiler dictionary in the host, but doesn’t add a header to the target. “:” goes ahead and adds the header to the target resident dictionary. Header was defined earlier in compiler.fs. The commented phrase with “talks” was used in the 8051 version of myforth where gforth was used for serial communications. This version uses minicom instead, but that could change in the future so it’s left here commented out.

```
47 :m exit
48   edge here 4 - = if \ lcall to ljmp
49     edge @-t $940e = if $940c edge !-t exit then
50   then
51   edge here 2 - = if \ rcall to rjmp
52     edge @-t $f000 and $d000 = if
53       edge @-t $1000 xor edge !-t exit then
54   then
55   $9508 ,-t m; \ ret
56 :m ; exit m;
```

“exit” is complicated by the ability to change a previous long or relative call to the corresponding jump for efficient tail recursion when possible, or compile an “ret” instruction otherwise. That’s enough of the assembler for now. We’ll see more of it later.



## “miscAVR.fs” Compiling Forth Code

```
40 \ variables on the target
41 variable ramp $100 ramp !
42 :m variable ( - adr) ramp @ 2 ramp +! m;
43 :m cvariable ( - adr) ramp @ 1 ramp +! m;
44 \ use as : this ( - a) variable # ;
```

“ramp” is short for RAM Pointer. It starts just above the stacks and grows upward. The word variable is definitely not standard. At compile (assembly) time it puts the address of the next available space in program memory on the stack and increments the pointer by two. In order to create a named variable you define a colon word and say “variable #” inside it. The # compiles the address as a literal.

```
50 :m begin ( - adr) hide here m;
51 :m again ( adr) hide rjmp, m;
52 :m until ( adr) hide 0 T adiw, rel $7f and breq, m;
53 :m -until ( adr) hide T' T' and, rel $7f and brpl, m;
```

These are some of the looping words. “begin” puts the address of the beginning of the loop on the stack at compile time for later resolution. The “hide” is there to tell the optimizer not to cross this loop boundary. “again” also tells the optimizer about the loop boundary and assembles a relative jump back to the address left by “begin”. We assume that loops will be small enough for relative jumps. “until” adds 0 to the 16 bit top of stack in order to set flags, then does a relative branch if equal to zero. Similarly “-until” sets flags and branches if the sign bit is set. Note that until does not pop the data stack. This is not it standard, but it’s how colorForth and arrayForth work.

```
55 \ loop until bit is set
56 :m /until ( bit) hide clr? again m;
57 \ loop until bit is clear
58 :m \until ( bit) hide set? again m;
```

Being a micro-controller, there are instructions for branching on the state of a port bit. “/until” loops until a bit is set. Think of / as a rising edge. “\until” loops until the bit is clear.

```
60 \ forward ... resolve ( long jump)
61 :m forward ( - adr) begin dup ljmp, m;
62 :m resolve ( adr) begin >r org r@ ljmp, r> org m;
63
64 \ ahead ... then ( short relative jump)
65 :m ahead ( - adr) begin dup rel $7f and rjmp, m;
66 :m then ( adr) ( here) begin >r dup org r@ rel $7f and
67   3 lshift over @-t $fc07 and or swap !-t r> org m;
```

“ahead then” is a fairly common way to jump forward unconditionally. It’s a relative jump, because “then” is designed to resolve a relative forward jump. Sometimes you need a longer forward jump and “forward resolve” does that. I made it up, not standard at all.

```
69 \ each of the following matches up with "then" for a short relative jump
70 :m if ( - adr) 0 T adiw, ( here) begin dup rel $7f and breq, m;
```

```

71 :m -if ( - adr) T' T' and, ( here) begin dup rel $7f and brpl, m;
72 :m while ( a - a' a) if [ swap ] m;
73 :m -while ( a - a' a) -if [ swap ] m;
74 :m repeat ( a a') again then m;
75 :m /if ( - adr) hide clr? ahead m;
76 :m \if ( - adr) hide set? ahead m;

```

These are the various flavors of “if”, including “while” and repeat is here because it goes with “while” and “-while”. Again, these conditionals do not pop the stack.

```

78 \ stack
79 :m dup T' -stx, T -stx, m; \ 2 or 0
80 :m ?dup \ removes redundant "drop dup" pairs
81     edge here [ 4 - - if ] dup [ exit then ]
82     edge @-t $918d = edge 2 + @-t $919d = [ and if ]
83     -4 allot [ exit then ] hint dup m;

```

“dup” is a macro, meaning it gets compiled in line rather than being called. It’s so short and so common that this seems a good bet. Many of the primitives are like this. “?dup” is not the standard version you might already know. This version optimizes code where a drop is followed by a dup, which is redundant in the context of this compiler. The user is not meant to use “?dup”, only the compiler does behind the scenes. This was also borrowed from colorForth.

```

85 :m drop hint T ldx+, T' ldx+, m; \ 2 or 0
86 :m nip N ldx+, N' ldx+, m; \ 2
87 :m |swap nip dup N T movw, m; \ 5
88 :m |over nip N' -stx, N -stx, dup N T movw, m; \ 7
89 :m push T' push, T push, drop m; \ 2 or 4
90 :m pop ?dup T pop, T' pop, m; \ 2 or 4

```

This is the stack manipulator group. The top of the data stack is cached in the T, T’ register pair. The N, N’ register pair holds the second item on the stack temporarily for binary operations. “swap” and “over” are too long to be inlined except maybe in special circumstances. Note the comments at the end of each word telling how many words of program memory they consume. Also note that “push” and “pop” take the place of the standard “>r” and “r>”. These names are borrowed from arrayForth.

```

92 \ binary
93 :m |+ nip N T add, N' T' adc, m; \ 4
94 :m +' nip N T adc, N' T' adc, m; \ 4
95 :m |and nip N T and, N' T' and, m; \ 4
96 :m |xor nip N T xor, N' T' xor, m; \ 4
97 :m |or nip N T or, N' T' or, m; \ 4
98 :m |- T N movw, drop N T sub, N' T' sbc, m; \ 5

```

All of the binary operators take too much space to be inlined. They have a bar in front in case you do want to inline them at some point. You’ll see later that calls are defined for each before your application is compiled. “+” is a special case. It’s the “add with carry” operator for extended precision arithmetic.

```

100 \ hand optimization
101 :m #+ ( n) [ dup 0 64 within 0= abort" number out of range" ]
102     T adiw, m; \ 1
103 :m #- ( n) [ dup 0 64 within 0= abort" number out of range" ]
104     T sbiw, m; \ 1

```

Since the AVR has instructions for adding short literals we have these two words that allow optimizing by hand. You can add or subtract a number in the range 0-63 directly to the top of stack in a single instruction. You just need to remember and to notice when opportunity knocks.

```

106 \ unary
107 :m invert T com, T' com, m; \ 2
108 :m |negate invert 1 #+ m; \ 3
109 :m 2* T T add, T' T' adc, m; \ 2
110 :m |2/ 7 T' bst, T' ror, T ror, 7 T' bld, m; \ 4

```

These unary operators each act on the T, T' register pair. Note. “bst,” instruction stores a bit, in this case the sign bit, in the flags register as T (for true?). “bld,” loads that T bit from the flags register to another register, in this case our T or top of stack register.

```

112 \ memory (A=y)
113 :m a! T Y movw, drop m; \ 3
114 :m a ?dup Y T movw, m; \ 3 or 1
115 :m @+ ?dup T ldy+, T' ldy+, m; \ 2 or 4
116 :m c@+ ?dup T ldy+, 0 T' ldi, m; \ 2 or 4
117 :m !+ T sty+, T' sty+, drop m; \ 2 or 4
118 :m c!+ T sty+, drop m; \ 1 or 3
119 \ 16 bit special function registers want
120 \ high byte *written* first but *read* last
121 :m |@ T Y movw, T ldy+, T' ldy+, m; \ 3
122 :m |! T Y movw, drop 2 Y adiw, T' -sty, T -sty, drop m; \ 8
123 :m |c@ T Y movw, T ldy+, 0 T' ldi, m; \ 3
124 :m |c! T Y movw, drop T sty+, drop m; \ 6

```

The memory operators group. They all act in internal RAM, where the variables are. The Y register is used as our address register. Thus the words “a!” and “a” which were borrowed from arrayForth. “a!” pops the top of stack into a, also known as Y. “a” pushes the value of the a register onto the stack. Each of the operators with a + sign on the end relies on your having already loaded the a register. “@”, “!”, “c@” and “c!” take an address off the stack and put it into the a register. The “+” operators each post increment the address register appropriately.

```

126 \ literal
127 :m # ( n) ?dup [ dup $ff and ] T ldi, \ 2 or 4
128     [ 8 rshift $ff and ] T' ldi, m;
129 :m ~# ( n) host invert target # m; \ 2 or 4

```

This being an assembler, we use an operator, #, to assemble a literal. “~#” inverts the literal at compile time, which is handy for making bit masks to be anded for clearing bits.

```

131 \ counted loop, be careful about using the Z register inside!
132 \ 10 for counts from 10 down to 1 in Z (R), but i shows the index
133 \ as 9 down to 0. r@ gets the unmodified index, 10 to 1,
134 \ or whatever else may be in Z.
135 :m for ( - adr) hide
136     Z' push, Z push, T Z movw, drop begin m; \ 5 (once)
137 :m next ( adr) 1 Z sbiw, [ rel $7f and ] brne, \ 2 (inside loop)
138     Z pop, Z' pop, hide m; \ 2 (at finish)
139 :m r@ ( - n) ?dup Z T movw, m;
140 :m i ( - n) r@ 1 T sbiw, m;

```

The AVR has three register pairs that can act like 16 bit registers. I wish there were (at least) one more. The Z register does double duty. It's used to cache the loop counter in for, next loops. It's also the only register that can index into program flash for reading. This doesn't come up too often, but it can be a real pain when it does. Unfortunately you can't easily read from program memory inside a for, next loop. Maybe we'll come up with a better way of doing this someday. Note that a for loop always executes at least once, counting down to and including zero. So "4 for i . next" shows "4 3 2 1 0".

```

142 \ 32 bit result in 2,3,4,5
143 :m |16*16=32
144     nip 20 20 xor, \ multiply T,T' N,N'
145     T' N' mul, 0 4 mov, 1 5 mov,
146     T N mul, 0 2 mov, 1 3 mov,
147     T N' mul, 0 3 add, 1 4 adc, 20 5 adc,
148     T' N mul, 0 3 add, 1 4 adc, 20 5 adc, m;

```

The "mul," operator helps us implement a fairly efficient 16x16 multiply. This is made into the usual multiply ops later.

```

150 \ dividend in 2,3,4,5 as left by 16*16=32 ; divisor in T
151 \ remainder in 4,5 ; quotient in 2,3
152 \ zero in 8 ; counter in N ; bit in 7
153 :m |32/16=16,16
154     N' N' xor, \ preload zero for comparison later
155     $10 N ldi, \ loop counter in N
156     begin
157         6 6 xor, 2 2 add, 3 3 adc,
158         4 4 adc, 5 5 adc, 6 6 adc, \ shift
159         T 4 cp, T' 5 cpc, N' 6 cpc, \ trial subtraction
160         3 brcs, 2 inc, T 4 sub, T' 5 sbc, \ actual subtraction
161         N dec, 2 breq, ljmp, m;

```

This is binary long division, used later for /, \*/, et cetera.

```

163 \ hand optimizing
164 :m apush Y' push, Y push, m;
165 :m apop Y pop, Y' pop, m;
166 :m zpush Z' push, Z push, m;
167 :m zpop Z pop, Z' pop, m;

```

It is often convenient to push or pop the Y or Z registers to and from the return stack. These words do so efficiently.

## Back to job.fs

It seemed as though one might want to be able to customize the stacks for some applications, so that happens here in the job file.

```
70 :m init-stacks
71   [ r0 dup 8 rshift $ff and ] T ldi, T SPH out,
72   T ldi, T SPL out, \ init return stack
73   [ s0 dup 8 rshift $ff and ] X' ldi, X ldi, m; \ init data stack
74
75 \ Default interrupt vector
76 0 org 0 ljmp, \ reset vector
77 0 ljmp, \ $04
78 0 ljmp, \ $08
79 0 ljmp, \ $0c
80 0 ljmp, \ $10
81 0 ljmp, \ $14
82 0 ljmp, \ $18
83 0 ljmp, \ $1c
84 0 ljmp, \ $20
85 0 ljmp, \ $24
86 0 ljmp, \ $28
87 0 ljmp, \ $2c
88 0 ljmp, \ $30
89 0 ljmp, \ $34
90 0 ljmp, \ $38
91 0 ljmp, \ $3c
92 0 ljmp, \ $40
93 0 ljmp, \ $44
94 0 ljmp, \ $48
95 0 ljmp, \ $4c
96 0 ljmp, \ $50
97 0 ljmp, \ $54
98 0 ljmp, \ $58
99 0 ljmp, \ $5c
100 0 ljmp, \ $60
101 0 ljmp, \ $64
```

I also explicitly initialize the interrupt vectors to point to the reset vector at address 0. I still like to stuff the vectors using the word “interrupt”.

## “primitives.fs”

```
24 \ primitive calls
25 : swap |swap ;
26 : over |over ;
27 : + |+ ;
28 : and |and ;
29 : xor |xor ;
30 : or |or ;
31 : - |- ;
32 : negate |negate ;
33 : 2/ |2/ ;
```

```

34 : @ |@ ;
35 : ! |! ;
36 : c@ |c@ ;
37 : c! |c! ;

```

As mentioned earlier, though many of the primitives are small enough to be inlined, some commonly used primitives are just too long, so they're compiled as callable words here. If you really feel the need to inline them, just use the versions with | in front. These are the first bits of code to occupy memory by the way, after the interrupt vectors.

## math.fs All the Usual, Plus Signed 14 Bit Fractions

```

24 -: 16*16=32 |16*16=32 ; \ basic multiplication
25 : um* ( u1 u2 - ud) 16*16=32
26   3 -stx, 2 -stx, 4 T mov, 5 T' mov, ;
27
28 -: 32/16=16,16 |32/16=16,16 ;
29 \ put dividend in 2,3,4,5 as if by 16*16=32
30 \ leave divisor in T,T'
31 \ get remainder from 4,5 quotient from 2,3
32 : um/mod
33   4 ldx+, 5 ldx+, 2 ldx+, 3 ldx+,
34   32/16=16,16 5 -stx, 4 -stx, 2 T mov, 3 T' mov, ;
35 : u/mod ( u1 u2 - rem quo) push 0 # pop um/mod ;

```

First we make those multiplication and division primitives into calls. We're going to use them each several times. Then come the unsigned primitives we're probably you used to, "um\*" and "um/mod", with "u/mod" as well. "um\*" calls the primitive "16\*16=32" and then pushes the result onto the stack in the correct order. "um/mod" has to first load registers 2, 3, 4 and 5 in correct order before calling the division primitive "32/16=16,16" then moves the result back to the stack. We've chosen to use registers 2, 3, 4 and 5 for the math primitives, so don't use those for long term storage. "u/mod" is handy, especially since currently the display words only convert single precision numbers.

```

37 : abs ( n - n') -if negate then ;
38 : ?negate ( u n - n') -if drop negate ; then drop ;
39 : dnegate ( d1 - d2) swap invert swap invert 1 # 0 # \ fall through
40 : d+ ( d1 d2 - d3) push swap push + pop pop +' ;
41 : dabs ( d - +d) -if dnegate then ;
42 : s>d ( n - d) dup \ fall through into 0<
43 : 0< ( n1 - flag) -if drop -1 # ; then drop 0 # ;

```

Before defining signed division in "sm/rem" a few helper words are needed, mostly having to do with double precision numbers. "d+" shows how to use "+" for multiple precision addition. Note also the use of ";" inside a definition for an early exit. In myforth the ";" doesn't necessarily end a word.

```

45 : sm/rem ( d n - r q)
46   over push over over xor push \ save signs
47   push dabs pop abs \ everything positive
48   um/mod pop ?negate \ apply sign to quotient
49   swap pop ?negate swap ; \ apply sign to remainder

```

I've chosen "sm/rem" as the signed division primitive. I'm not convinced that floored division is better. If you'd prefer floored division these tools should allow you to define it yourself.

```

51 \ signed integers
52 : * 16*16=32 2 T mov, 3 T' mov, ;

```

This version of "\*" is slightly more efficient than calling "um\* drop", which is why the "16\*16=32" word exists.

```

54 : m* ( n1 n2 - d) over over xor invert push
55   push abs pop abs um* pop -if drop ; then
56   drop dnegate ;

```

This word is convenient in itself, but is also used in the definition of "\*/".

```

58 : /mod ( n1 n2 - rem quo) push s>d pop sm/rem ;
59 : mod ( n1 n2 - rem) /mod drop ;
60 : / ( n1 n2 - quo) /mod nip ;

```

These commonly used words could easily be commented out if you need the space and aren't using them in your program. They can be handy for interactive debugging though.

```

62 : */ ( n1 n2 n3 - n4) push m* pop sm/rem nip ;

```

"star-slash" was the reason for many of the previous words. What good is Forth without "\*/"?

```

64 \ signed fractions where +1=$4000
65 : *. 16*16=32
66   3 3 add, 4 4 adc, 5 5 adc,
67   3 3 add, 4 4 adc, 5 5 adc,
68   4 T mov, 5 T' mov, ;
69 : +1 $4000 # ;
70 : /. +1 swap */ ;
71 : >f 10000 # /. ;

```

These 14 bit fractional operators are great for things like trigonometry, and avoiding floating point in general.

## standalone.fs A Target Resident Text Interpreter

The standalone interpreter serves two purposes (at least). First it allows interactive debugging. It is also a source of many non-primitive words that can be useful in testing code. These include "key" and "emit" for serial communications and some number display words, as well as read access to program memory.

```

24 : last cvariable # ;
25 : base variable # ;
26 : hex $10 # base ! ;
27 : decimal $0a # base ! ;

```

Here's an example of using variable and cvariable. Define "last" as a normal colon word. "cvariable" puts the next available locate in RAM on the stack at compile time. "#" compiles that number as a literal. Same thing for "base", except variable allots two bytes of RAM instead of just one. "hex" and "decimal" will be useful for debugging later.

```

28 : = - \ falls through into 0=
29 : 0= if -1 # or then invert ;

```

"=" will be used in the interpreter, and "0=" comes with it for free, so why not? Note that you don't have to use ";" before defining another word with colon. Like arrayForth, myforth just falls through into the the next word. This can be a space saver.

```

30 : 2drop drop drop ;
31 : 2dup over over ;
32 : min 2dup swap
33 -: clip - -if push swap pop then drop drop ;
34 : 0max 0 # \ falls through into max
35 : max 2dup clip ;

```

"min" and "max" are needed in the interpreter and "0max", "clip", and "2drop" come along with them. I see that a chance was missed to let "clip" fall into "2drop", saving some memory. I'll fix that later. Note the "-:" before clip instead of ":". "clip" will be headless in the target image. It's useful in both "min" and "max". "min" falls through into it and "max" calls it. It doesn't seem like a very useful word otherwise, so we don't waste any head space for it.

```

37 : emit ( c)
38   apush 0 #
39   begin drop UCSR0A # c@ $20 # and until
40   drop UDR0 # c! apop ;
41 : key ( - c)
42   apush 0 #
43   begin drop UCSR0A # c@ $80 # and until
44   drop UDR0 # c@ dup last c! apop ;

```

"emit" and "key" are the serial interface. They use the hardware UART in the AVR via two special function registers. The variable "last" remembers the last key input. This is so that "ok" can decide whether to do a carriage return or not. I'll explain more about that later.

```

45 : type ( adr len) 0max if apush
46   swap a! for c@+ emit next apop ;
47   then 2drop ;
48 : space 32 # emit ;
49 : cr 13 # emit 10 # emit ;

```



“type” is defined in terms of “emit”. Note the use of “apush” and “apop” to save and restore the a register while using “c@+”. This is in case “type” gets used in a word that also needs the address register. “space” and “cr” show how to compile numbers like 32, 13 and 10.

```
50 -: (digit) -10 # + -if -39 # + then 97 # + ;
51 -: digit $0f # and (digit) emit ;
52 : 16/ 2/ 2/ 2/ 2/ ;
53 -: (h.) ( n) dup 16/ 16/ 16/ digit
54     dup 16/ 16/ digit dup 16/ digit digit ;
55 : h. ( n) (h.) space ;
56 : h.2 ( n) dup 16/ digit digit space ;
```

These words were defined when I thought there was no RAM to spare and maybe hadn’t added long division yet. It’s a way to display hex numbers without dividing, just shifting. It deserves to be rethought sometime.

```
57 -: sp@ ( - a) dup X T movw, ;
58 : depth ( - n) s0 # sp@ - 2/ 2 # - ;
```

This is how easy it is to mix assembler and Forth. In “sp@” “dup” makes room on the stack for the value of the stack pointer, and “X T movw,” copies the value from X to T in a single instruction.

```
59 -: ok last c@ BL # = if drop ; then drop
60     [ char o ] # emit [ char k ] # emit cr ;
```

Another example of some weird tradeoffs I made. I didn’t want to come up with string constants just to type “ok” at the end of a command, so I use “emit” instead. Note the use of [ and ] to be sure that “char” is happening on the host at compile time. “literal” is not needed because this is an assembler and there is no compile mode. We’re just interpreting on the host really, not “compiling”. Also this is where “last” comes in. When the last character typed before executing “ok” was a BL or space character we skip over the line that types “ok” and does a carriage return. One more thing. See that there’s a “drop” after the “if” and also after the “then”. This is because “if” doesn’t pop the stack, like arrayForth.

```
61 -: tib! ( c) apush tib # dup c@ 1 #+ over c! dup c@ + c! apop ;
```

“tib!” puts a character into the terminal input buffer and increments the character count. Since “c@” and “c!” are being used we save and restore the a register. It’s a pain but there just aren’t enough 16 bit registers!

```
62 -: huh? [ char ? ] # emit forward ( *)
63 -: ?stack depth -if huh? ; then drop ;
```

“huh?” is used to display a question mark and then abort. Since “abort” hasn’t been defined yet, we use “forward” to make an unconditional forward jump, to be resolved by “resolve” later when “abort” is being defined. “?stack” checks for stack underflow while interpreting.

```

65 \ Be careful with these, the Z register is dedicated
66 \ as a loop counter, and must be preserved.
67 -: c@p+ ( - c) dup lpm, 0 T' ldi, 0 T mov, 1 Z adiw, ;
68 -: c@p ( - c) c@p+ 1 Z sbiw, ;
69 -: @p+ ( - n) dup lpm, 1 Z adiw, 0 T mov,
70   lpm, 1 Z adiw, 0 T' mov, ;
71 :m p! ( a) T Z movw, drop m;
72 :m p ( - a) ?dup Z T movw, m;
73 :m #p! ( adr) [ dup $ff and ] Z ldi,
74   [ 8 rshift $ff and ] Z' ldi, m;

```

This is a group of words used to read strings out of program memory. I'm calling the Z register "p" here, like the p register in arrayForth. There are no store instructions since this is flash. Z is used as the index register, making this difficult to use in a counted loop. I see looking at the data book that there is also an "lpm" instruction that increments Z, so I'll have to rethink this too. It would seem that Z contains a byte address for "lpm". "#p!" loads the Z or p register directly, by-passing the data stack.

```

76 target
77 here constant dict \ patch location of dictionary later
78 -: dictionary 0 #p! ;

```

Now we get to the target resident dictionary. The host constant "dict" (there are no target constants, just literals) marks the place where the dictionary pointer will be patched in later. The word "dictionary" loads that address into p, AKA the Z register.

```

79 -: .word ( c) zpush c@p+ begin c@p+ emit 1 #- while repeat
80   drop space zpop c@p+ p + dup 1 # and + 2 # + p! ;
81 : words dictionary begin c@p while
82   drop cr .word key 13 # = if drop ; then drop repeat drop ;

```

The headless ".word" prints out the characters of the word name that the p register is pointing to, leaving p pointing to the name of the next word. "words" uses "dictionary" to point at the first word in the list then loops printing words until top of stack counts down to zero. After each word is printed it waits for a key press not equal to the carriage return before printing the next one. A carriage return or reaching the end of the list will cause an exit to the interpreter.

```

106 : quit query interpret ?stack ok quit ;

```

Let's peek ahead and see where the next few words are leading. "quit" is the main loop of the interpreter. "query" is for entering a word at the terminal. "interpret" looks the word up in the dictionary and executes it if found, or tries to see it as a number. "?stack" aborts if the stack has underflowed. "ok" prints "ok" and does a carriage return if we've reached the be end of a line, and "quit;" jumps back to quit in an endless loop. Note that we interpret a are word at a time rather than a line at a time. This was intended to save memory.

Now we'll look at what leads up to those words.

```

83 -: word tib # a! c@p 1 #+
84   begin c@p+ c@+ xor if nip ; then drop 1 #- while repeat ;

```

“word” compares the string that p points to in the dictionary against the word in the terminal input buffer. It xor’s corresponding pairs of characters until they differ or they run out. If they differ the stack contains something other than 0. If they’re still the same or they’ve run out then the stack will contain a 0. “word” is used by “find”.

```

85 -: find zpush c@p if drop word if
86   drop zpop c@p+ p + dup 1 # and + 2 # + p! find ; then
87   drop zpop c@p+ p + dup 1 # and + p! @p+ ; then
88   zpop drop 0 # ;

```

“find” first saves the p or Z register. If the first character is 0 we go to the last line, pop Z, drop the flag, and exit with a 0 on the stack, meaning we didn’t find anything. Seems there was already a zero on the stack so that was redundant. I’ll fix that later. If “c@p” says there is a word there, then we run “word” and learn whether or not there’s a match. If not we find the next word in the list by adding the count to the base address + 2 (execution address), jump back to “find” (tail recursion) and continue searching. If “word” gave us a match, then we calculate the address that contains the execution address for the match, fetch the contents of that address and leave it on the stack. So “find” ends up with either the execution address of the word found or a zero on the stack.

```

89 : execute ( a) T push, T' push, drop ;

```

Having found the address to be executed and left it on the stack, the word “execute” can execute it by pushing it onto the return stack and returning.

```

90 -: digit? ( c - c' flag) [ char 0 ] # - -if -1 # ; then
91   10 # - -if 39 # + then 29 # -
92   apush base @ - -if base @ + 0 # apop ; then -1 # apop ;

```

In the case that we didn’t find the word in the dictionary, we still need to see if it’s a number. “digit?” checks to be sure a character is a legitimate digit in the current base, leaving a flag.

```

93 -: +number? tib # a! 0 # c@+ \ fall through
94 -: -number?
95   for apush base @ * apop c@+ digit? if
96   push push push drop 0 # pop pop pop then drop + next swap ;
97 -: number? -1 # [ tib 1 + ] # c@ 45 # xor if drop +number? ; then
98   tib # a! c@+ 1 #- c@+ drop -number? push negate pop ;

```

“number?” checks the first character in tib and if it’s not a minus sign jumps straight to “+number?” to see if it can be a positive number. If the first character is a minus sign then we skip over that character and jump to “-number?”, evaluate it as a positive number and negate it at the end.

```

99 -: interpret apush dictionary find if apop execute ; then
100   drop number? if drop apop ; then huh?

```

Now there's enough support to define "interpret". After a word has been typed into the terminal input buffer (query is next) we save the a register and search the dictionary. If the word was found, pop the a register and execute it, done. If not a word, see if it's a number. If so leave it on the stack, restore the a register, done. Otherwise call "huh?" which prints a question mark and aborts, initializing the stacks and starting over.

```

101 -: query 0 # tib # ! \ fall through
102 -: back key dup 8 # = if 2drop cr query ; then
103   drop BL # max echo BL # xor if BL # xor tib! back ; then
104   drop apush tib # c@ if drop apop ; then drop apop ok query ;

```

"query" is a very simplistic editor based on the one in colorForth. I may want to revisit this and see how hard it would be to have an editor with actual backspaces. But for now, this is it. "query" puts a zero count into tib and accepts characters into tib incrementing the count. If a backspace character, ascii 8, is encountered clear the stack, do a carriage return and start over at query. No editing allowed. If the character is not a space then store it in tib and jump to back to continue accumulating characters. If the character was a space (or any whitespace character) then if the count is not zero this was a word so we store a and exit. If it was zero then all that was typed was a single space. We say ok and jump back to query to wait for a word.

```

105 : abort ( *) resolve cr init-stacks
106 : quit query interpret ?stack ok quit ;

```

Now all the words in "quit" have been defined. "abort" resolves that forward jump from "huh?", does a carriage return, initializes the stacks and falls through into "quit", which reads a word in and interprets it, checking for stack underflow and looping. That's enough for a standalone interpreter. Still, it's nice to be able to display decimal numbers and dump the stack with ".s" so there's more.

```

109 -: sign -if negate 45 # emit ; then ;

```

If the number on the stack is negative, negate it and emit a minus sign.

```

112 : (u.) ( u) apush 0 # first ! -1 # swap
113   begin base @ u/mod while repeat drop
114   begin digit -until drop ( space) apop ;
115 : u. ( u) (u.) space ;

```

"(u.)" saves a, puts a -1 on the stack as a marker, and extracts digits by dividing by the current base until nothing is left. All the digits of the number are on the stack with a -1 marking the end of the list. Then it prints digits until it gets to the -1 marker and restores a. "u." does the same and then prints a space afterward.

```

116 : .f ( f) apush sign 10000 # *.
117     10000 # u/mod digit 46 # emit 1000 # u/mod digit
118     100 # u/mod digit 10 # u/mod digit digit space ;

```

“f” displays a 14 bit fraction by printing the sign if negative, printing the whole number part followed by a decimal point, then four digits of fraction. It’s assumed that we want to see this as a decimal number.

```

119 \ . is unsigned for hex, signed otherwise
120 : . apush base @ apop $10 # xor if drop sign dup then drop u. ;

```

“.” cleverly shows a decimal number as signed but a hex number as unsigned.

```

121 : .s apush [ s0 4 - ] # a! depth dup (h.) 62 # emit space
122     if dup 6 # min for @+ . 4 Y sbiw, next then drop apop ;
123 : ? @ . ;

```

“s” shows the depth of the stack no matter what, then uses “.” to show whatever is on the stack in the current base, but only up to the top six numbers. “?” will fetch from a variable and display it using “.”. That’s the current state of the standalone interpreter.

## Back to job.fs again

This is where the main application gets compiled, in the file named “main.fs”. Be sure to include a word named “go” that will be the main loop of your app. Even if there is no “go” though, you can execute “abort” instead of “go” to run the interpreter. In fact that’s what you should do until the program has been debugged.

At this point in job.fs a macro is defined to initialize the serial port.

```

109 :m init-serial
110     DDRD Y ldi, 2 T ldi, T sty, \ TX0 is output
111     UCSR0A Y ldi, 0 Y' ldi, \ 8N1
112     $20 T ldi, T sty+, \ ready to transmit or receive
113     $18 T ldi, T sty+,
114     6 T ldi, T sty+,
115     UBRR0L Y ldi, 0 Y' ldi, \ 9600 baud
116     103 T ldi, T sty+,
117     0 T ldi, T sty+, m;

```

This code sets up as 9600 baud, N81. Look in the data book to see what needs to change for different baud rates. I don’t remember without looking it up.

```

119 \ Uncomment if you haven't already defined this word,
120 \ in timer.fs for example.
121 : init-interrupt ;

```

This is a good place to initialize interrupts if there are any. In this case there aren’t so the word does nothing.

```

123 target
124 \ some macros are made subroutines just for interactive testing
125 include ../interactive.fs

```

## interactive.fs

A number of primitives have only been available as macros up to this point. If we want to use them interactively then they need to be compiled here as calls with target heads.

```

3  :m |dup dup m;
4  :m |drop drop m;
5  :m |nip nip m;
6  :m |invert invert m;
7  \ :m |negate negate m;
8  :m |2* 2* m;
9  \ :m |2/ 2/ m;
10
11 \ :m |@ @ m;
12 :m |a a m;
13 :m |a! a! m;
14 :m |@+ @+ m;
15 :m |!+ !+ m;
16 :m |c@+ c@+ m;
17 :m |c!+ c!+ m;
18
19 target
20 : dup |dup ;
21 : drop |drop ;
22 : nip |nip ;
23 : invert |invert ;
24 \ : negate |negate ;
25 : 2* |2* ;
26 \ : 2/ |2/ ;
27
28 \ : @ |@ ;
29 : a |a ;
30 : a! |a! ;
31 : @+ |@+ ;
32 : !+ |!+ ;
33 : c@+ |c@+ ;
34 : c!+ |c!+ ;

```

Since myforth is recursive we need to make an alias for each before defining calls through the aliases. Now these words can be used for debugging in the interpreter.

```

126 \ Here you decide whether to quit or go, meaning debug or run
127 : cold entry cli, init-serial 10 # base ! init-interrupt abort ;
128 \ : cold entry cli, init-serial init-stacks 10 # base ! init-interrupt go
;

```

One of these lines should be commented out. The one ending in “abort” runs the interpreter. The one ending in “go” turnkeys your application. “entry” stuffs the address of “cold” into the reset vector at

address 0. The “cli,” instruction clears the interrupt bit, disabling interrupts. The serial port is initialized. Base is set to decimal. Interrupts, if there are any, are initialized, and we jump to “abort” to start interpreting.

```
129 here [ dup ] dict org #p! org headers \ tack headers on end
```

This line stuffs the address of the end of the dictionary into the place we marked with “dict” earlier. Then “headers” moves the target headers down to the end of the dictionary.

```
131 host : .stack depth if >red then .s >black cr ;
132 report
133 save \ chip.bin, avrdude handles binary files just fine.
134 host .( Host stack= ) .stack
```

“.stack” is defined for the host to show the stack in eye-catching red if we made a mistake and left something on the host stack. “report” announces the size of the image. “save” saves the image to a file named “chip.bin”. Finally we run “.stack” to see if we made any obvious errors.

That’s myforth-arduino for the AVR328 chip.